Politechnika Warszawska

WYDZIAŁ ELEKTRONIKI
I TECHNIK INFORMACYJNYCH



Instytut Radioelektroniki i Technik Multimedialnych

Praca dyplomowa inżynierska

na kierunku Telekomunikacja
w specjalności Radiokomunikacja i Techniki Multimedialne

Realizacja sprzętowo-programowa algorytmu kryptograficznego z rodziny Argon2

Bartosz Zabołotny
Numer albumu 293295

promotor
dr hab. inż. Mariusz Rawski

WARSZAWA 2021

Realizacja sprzętowo-programowa algorytmu kryptograficznego z rodziny Argon2

Streszczenie. W pracy przedstawiono koncepcję, implementację oraz metody weryfikacji realizacji sprzętowo-programowej algorytmu Argon2d z wykorzystaniem układu SoC FPGA. Ze względu na wysoką odporność na próby przyspieszania obliczeń przy pomocy układów GPU, FPGA oraz ASIC, algorytm wykorzystywany jest w części kryptowalut oraz do wyznaczania skrótów haseł.

Przedstawiono proces weryfikacji koprocesora z wykorzystaniem środowiska *CocoTB* oraz przygotowania obrazu systemu przy pomocy narzędzia *Buildroot.* Zaprezentowano i omówiono wyniki porównania realizacji czysto programowej z wynikami uzyskanymi z użyciem akceleratora oraz opisanymi w literaturze rozwiązaniami.

Słowa kluczowe: SoC, FPGA, akceleracja, Open Source, CocoTB, Buildroot

Hardware-software implementation of the Argon2 cryptographic algorithm

Abstract. The thesis presents the concept, implementation, and verification methods of the hardware-software version of the Argon2d algorithm executed in SoC FPGA. This algorithm is used in some cryptocurrencies and password hashing due to high GPU-, FPGA- and ASIC-resistance.

The process of coprocessor verification using the *CocoTB* framework and the system image's preparation using the *Buildroot* tool was presented. The pure software and accelerator-based implementations are compared with the solutions described in the literature. The results are shown and discussed.

Keywords: SoC, FPGA, acceleration, Open Source, CocoTB, Buildroot

załącznik nr 3 do zarządzenia
nr 28 /2016 Rektora PW



Politechnika Warszawska

…………………………..
miejscowość i data

……………………………..
imię i nazwisko studenta

……………………………..
numer albumu

……………………….………
kierunek studiów

OŚWIADCZENIE

Świadomy/-a odpowiedzialności karnej za składanie fałszywych zeznań oświadczam, że niniejsza praca dyplomowa została napisana przeze mnie samodzielnie, pod opieką kierującego pracą dyplomową.

Jednocześnie oświadczam, że:

- niniejsza praca dyplomowa nie narusza praw autorskich w rozumieniu ustawy z dnia 4 lutego 1994 roku o prawie autorskim i prawach pokrewnych (Dz.U. z 2006 r. Nr 90, poz. 631 z późn. zm.) oraz dóbr osobistych chronionych prawem cywilnym,
- niniejsza praca dyplomowa nie zawiera danych i informacji, które uzyskałem/-am w sposób niedozwolony,
- niniejsza praca dyplomowa nie była wcześniej podstawą żadnej innej urzędowej procedury związanej z nadawaniem dyplomów lub tytułów zawodowych,
- wszystkie informacje umieszczone w niniejszej pracy, uzyskane ze źródeł pisanych i elektronicznych, zostały udokumentowane w wykazie literatury odpowiednimi odnośnikami,
- znam regulacje prawne Politechniki Warszawskiej w sprawie zarządzania prawami autorskimi i prawami pokrewnymi, prawami własności przemysłowej oraz zasadami komercjalizacji.

Oświadczam, że treść pracy dyplomowej w wersji drukowanej, treść pracy dyplomowej zawartej na nośniku elektronicznym (płycie kompaktowej) oraz treść pracy dyplomowej w module APD systemu USOS są identyczne.

....
czytelny podpis studenta

Spis treści

1. Wstęp

Funkcje skrótu, nazywane inaczej funkcjami haszującymi, umożliwiają przyporządkowanie dowolnej liczbie pewną wartość o zadanym rozmiarze - tzw. *skrót* albo *hash*. Wartości skrótów powinny mieć charakter pseudolosowy, a dowolna, nieznaczna zmiana wartości wejściowej powinna skutkować znaczną zmianą wartości skrótu. Cechy te powodują, że obecnie funkcje te mają szereg różnych zastosowań.

Algorytm MD5 [1] generuje 128-bitowy skrót dla dowolnie długiego ciągu danych. Pomimo że od 2011 roku jest odradzany do zastosowań kryptologicznych [2], obecnie jest dalej często stosowany do weryfikacji spójności zbiorów danych. W przypadku transmisji danych, niezależnie od medium, możliwe jest wystąpienie losowych przekłamań. Znając skrót oryginalnego zbioru danych można wyznaczyć skrót odebranych danych, a następnie sprawdzić czy obydwa skróty są sobie równe. W analogiczny sposób można detekować przekłamania przy odczycie z pamięci flash, wynikające z losowych błędów utraty danych, gdy liczba przekłamanych bitów przekracza możliwości kodu korekcyjnego.

Systemy kontroli wersji, poza weryfikacją spójności repozytoriów, wykorzystują skróty w celu identyfikacji zbiorów danych. Do niedawna jeden z najpopularniejszych takich systemów - Git wykorzystywał w tym celu algorytm SHA-1 [3], a obecnie zastępuje go SHA-256 [4].

W większości języków programowania implementowane są tablice mieszające, nazywane nieraz mapami albo słownikami. Budowane są jako tablica, gdzie dana jest umieszczana pod indeksem równym wartości skrótu klucza. Pod warunkiem zagwarantowania braku kolizji skrótów i dobrania odpowiedniej do haszowanych danych funkcji skrótu, umożliwiają one wyszukiwanie, wstawianie oraz usuwanie danych ze złożonością $\Theta(1)$.

Funkcje, które poza odpornością na kolizje są dodatkowo uznawane za jednokierunkowe, tzn. znajomość hashu praktycznie nie niesie żadnej informacji o zbiorze danych wejściowych, stosowane są w kryptografii jako podpis cyfrowy, tudzież umożliwiają bezpieczne przechowywanie haseł [5]. Stały się również bazą do stworzenia kryptowalut, czyli cyfrowego systemu płatności w oparciu o (zazwyczaj) umowne jednostki.

Gwałtowny rozwój architektur umożliwiających wysoki stopień zrównoleglania obliczeń jak np. karty graficzne (GPU), układy FPGA czy nawet dedykowane układy układy ASIC, umożliwił w ostatnich latach znaczną redukcję kosztów łamania haseł. Spowodowało to potrzebę projektowania nowych algorytmów do wyznaczania skrótów haseł, dla których koszt prowadzenia obliczeń na procesorze architektury x86 czy ARM będzie porównywalny albo niższy, niż w przypadku wymienionych wcześniej układów. W 2013 roku ogłoszono konkurs *Password Hashing Competition* (**PHC**) na nowy algorytm haszowania haseł [6], a w lipcu 2015 roku wybrano jako zwycięskie rozwiązanie rodzinę algorytmów Argon2 [7].

Wariant **Argon2d** oferuje najwyższy poziom odporności na ataki z kompromisem czasu i pamięci, za cenę wrażliwości na ataki typu side-channel. To spowodowało, że jest

on chętnie stosowany w wielu kryptowalutach jako mechanizm dowodu wykonania pracy **PoW** (*Proof of work*), jako algorytm który praktycznie uniemożliwia prowadzenie efektywnych obliczeń poza CPU. Rozwój rynku kryptowalut skutkuje postępującą optymalizacją urządzeń do prowadzenia obliczeń na rzecz weryfikacji transakcji (*kopania*).

Podstawowym założeniem, na którym opierane są kryptowaluty jest decentralizacja sieci, jako gwarant zaufania i czynnik znacznie utrudniający ataki klasy 51% [8]. Bitcoin [9] będący obecnie jedną z największych kryptowalut był pierwotnie projektowany przy założeniu, że obliczenia będą prowadzone na procesorach domowych komputerów osobistych, w związku z czym większość mocy obliczeniowej sieci będzie rozproszona na dużą liczbę niezależnych węzłów. Wraz ze wzrostem popularności każdej kryptowaluty wykorzystującej mechanizm PoW rośnie ryzyko, że większość mocy obliczeniowej danej waluty zostanie skupiona w obrębie pojedynczych węzłów prowadzących obliczenia przy wykorzystaniu wyspecjalizowanych układów ASIC, tym samym praktycznie uniemożliwiając efektywne kopanie małym graczom, których nie stać na inwestycję w porównywalnie wydajny sprzęt oraz potencjalnie prowadząc do niepożądanej centralizacji waluty [10].

W latach 2011-2012 zaczęły się pojawiać na rynku pierwsze układy SoC FPGA (*System on Chip*) oferujące połączone w pojedynczym chipie układ programowalny oraz procesor ARM Cortex-A9 [11] [12] [13]. Nowa klasa układów umożliwiła projektowanie hybrydowych rozwiązań, gdzie dzięki bezpośredniemu połączeniu części programowalnej z CPU można efektywnie projektować koprocesory stosowane w roli akceleratorów. Dzięki temu można łatwo przenieść fragmenty, albo całe algorytmy, które się dobrze zrównoleglają, do wyspecjalizowanego koprocesora w części programowalnej.

1.1. Cel pracy

Celem pracy jest:

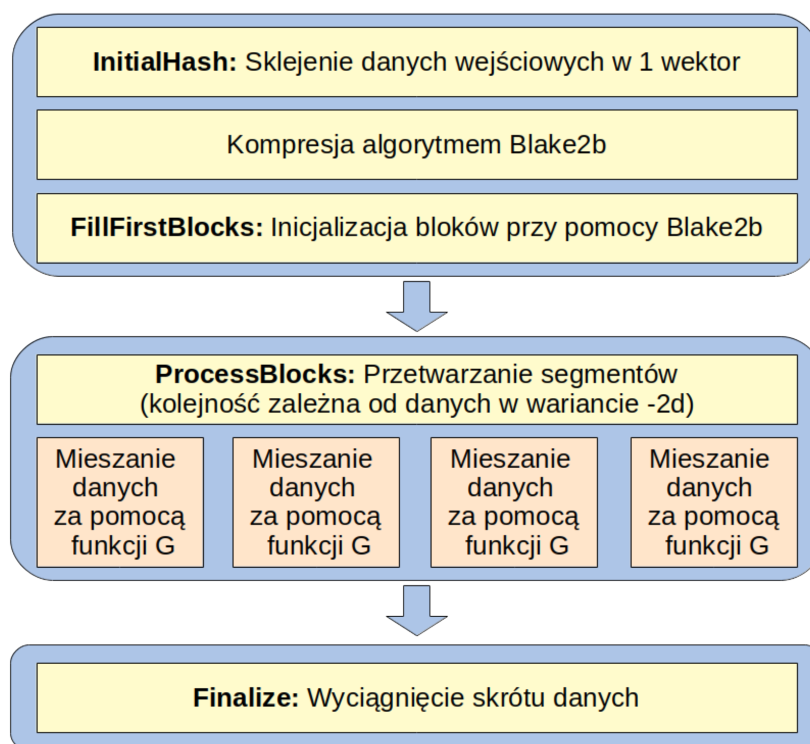
- zaprojektowanie implementacji sprzętowo-programowej algorytmu Argon2d,
- opisanie metod weryfikacji części sprzętowej systemu oparte na symulacji z wykorzystaniem zestawu otwartoźródłowych narzędzi,
- przygotowanie porównania implementacji czysto programowej ze sprzętowo-programową oraz dodatkowo wariantami wykorzystującymi wbudowany w procesor blok wykonujący instrukcje SIMD - *Neon*,
- ocena efektywności obliczeń.

2. Stan wiedzy

2.1. Rodzina Argon2

Rodzina algorytmów Argon2 [7] [14] jest odpowiedzią na ciągły postęp w dziedzinie wyspecjalizowanego sprzętu, który można wykorzystać do łamania haseł. Od samego początku opracowywana była pod kątem optymalizacji obliczeń dla procesorów architektury *x86* spotykanych na co dzień w zwykłych komputerach, jednocześnie maksymalnie utrudniając przyspieszanie obliczeń w układach ASIC, FPGA czy GPU. A.Biryukov, D.Dinu i D.Khovarotovich przyjęli za kryterium oceny odporności algorytmu iloczyn powierzchnia-czas.

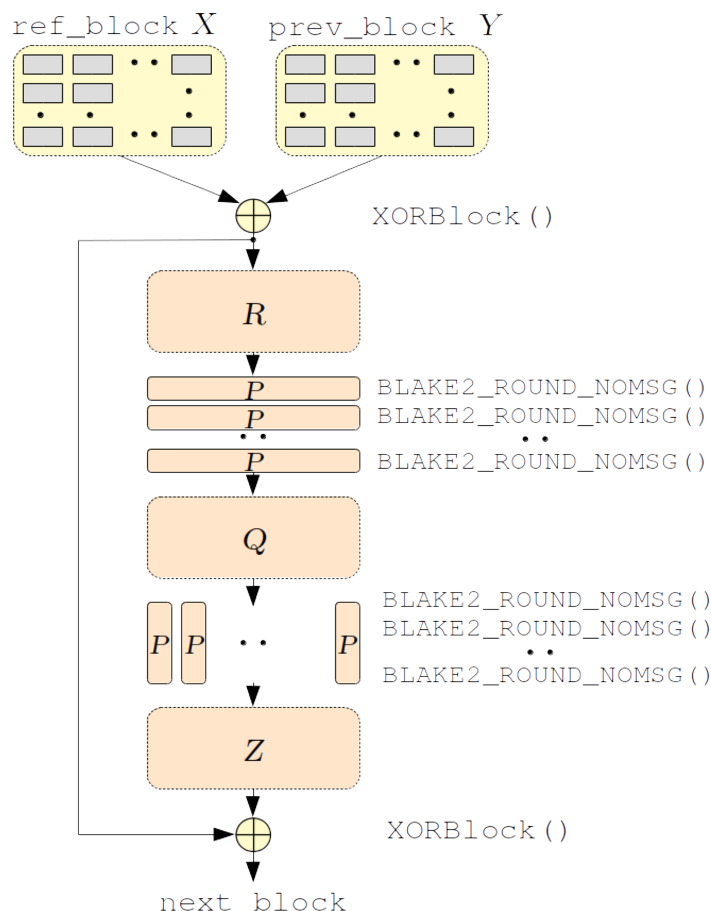
W repozytorium GitHub jednego z autorów zostały opublikowane różne wersje kodu algorytmu [15]. W pracy wykorzystano implementację referencyjną, tzn. bez optymalizacji dla architektury x86, w standardzie C99.



Rysunek 2.1. Uproszczony schemat algorytmu Argon2 dla parametru stopnia zrównoleglenia $p = 4$.

Algorytm można podzielić na 2 fazy: inicjalizacji bloków oraz przetwarzania segmentów. W fazie inicjalizacji wyznaczany jest skrót argumentów (rys.2.1: **InitialHash**), by potem wykorzystać tę wartość do inicjalizacji pierwszych bloków z użyciem algorytmu **Blake2b** (**FillFirstBlocks**). W fazie przetwarzania segmentów (**ProcessBlocks**) wykonywana jest główna część algorytmu - mieszanie danych. Polega ono na tworzeniu nowych 1024-bajtowych bloków poprzez wywołanie funkcji kompresji **G**, która zostanie opisana dalej, od poprzedniego bloku w segmencie oraz drugiego, referencyjnego bloku. Zależnie

od wariantu algorytmu, stosowane są różne sposoby wyznaczania indeksów bloku referencyjnego. W wariancie -2d wartości indeksów zależą od danych znajdujących się w poprzednim bloku, co miało na celu wymuszenie przechowywania kompletu wcześniejszych bloków w pamięci, jednocześnie podnosząc powierzchnię implementacji sprzętowych. Na końcu, w celu wyznaczenia finalnego bloku xorowane są ze sobą ostatnie bloki z kolejnych segmentów (**Finalize**).



Rysunek 2.2. Schemat funkcji kompresji G

Funkcja kompresji G, nazwana w kodzie referencyjnym `FillBlock`, została przedstawiona na rys.2.2. Przyjmuje jako argumenty 2 1024-bajtowe bloki danych, jeśli dane bloki są krótsze to zostają dopełnione z lewej strony zerami. Bazuje na zmodyfikowanej funkcji P z algorytmu Blake2b [16], dostępnej w kodzie referencyjnym jako `blake2b_compress` i w oryginalnym wariancie wykorzystywanej w procedurze *InitialHash*. Kluczową zmianą w funkcji P w celu utrudnienia implementacji ASIC było zastąpienie pewnych stałych iloczynami. W ramach pojedynczej rundy, blok dzielony jest na 8 wierszy lub kolumn, na których wykonywana jest pojedyncza, zmodyfikowana runda algorytmu Blake2b, której w kodzie referencyjnym algorytmu odpowiada makro `BLAKE2_ROUND_NOMSG`.

Na zmodyfikowaną rundę Blake2b składa się wielokrotne wywołanie funkcji *Mix*. Rozpatrując 1024-bitowy wektor danych przekazany jako argument jako macierz 4x4 słów

64-bitowych zapisaną kolejno wierszami, funkcja Mix jest najpierw wykonywana na każdej ćwiartce macierzy czytanej kolumnami, a następnie na każdej ćwiartce czytanej na skos. Zmodyfikowana funkcja $Mix(a,b,c,d)$, gdzie a,b,c,d są kolejnymi wyrazami macierzy, wygląda zatem w sposób następujący:

$$a := a + b + 2 \cdot a_{32} \cdot b_{32} \quad \textit{Nazywana w kodzie } \texttt{fBlaMka}$$
$$d := (d \oplus a) \ggg 32 \quad \textit{Nazywana w kodzie } \texttt{rotr64}$$
$$c := c + d + 2 \cdot c_{32} \cdot d_{32}$$
$$b := (b \oplus c) \ggg 24$$
$$a := a + b + 2 \cdot a_{32} \cdot b_{32}$$
$$d := (d \oplus a) \ggg 16$$
$$c := c + d + 2 \cdot c_{32} \cdot d_{32}$$
$$b := (b \oplus c) \ggg 63.$$

Przyjęto, że wszystkie sumy i iloczyny są liczone modulo 2^{64}; n_{32} odpowiada iloczynowi bitowemu $n \otimes 0xFFFF_FFFF$; a $\ggg$ jest operatorem obrotu bitowego w prawo. Fragment funkcji G wyróżniony kolorem ceglastym na rys.2.2 w dalszej części pracy nazywany będzie **funkcją mieszania**.

2.2. Podobne rozwiązania

W trakcie przygotowywania pracy, nie natrafiono na żadne rozwiązania, gdzie algorytm Argon2d próbowanoby uruchomić w układzie SoC FPGA. Nandakumar [17] zaprezentował w 2019 roku implementację wariantu Argon2id, będącą połączeniem wariantów -2i oraz -2d, przetestowaną na układzie FPGA z rodziny Xilinx Spartan-6. Niestety przedstawione tam wyniki są niespójne. Autor zadeklarował, że otrzymano z pewnego, niezamieszczonego w publikacji równania przepustowość wynoszącą 603,4Mbit/s. Brakuje informacji w publikacji na temat tego, w jaki sposób zdefiniowano przepustowość. Rozmiar wyznaczanego skrótu - w wynikach podawany jest 128 bitów, natomiast w abstrakcie - 256 bitów. Jednocześnie częstotliwość zegara wynosiła 63,56MHz, a parametr stopnia zrównoleglenia $p = 4$. Latencja zdefiniowana jako liczba cykli zegara potrzebna do wyznaczenia hasła wynosiła 64. Rozmiar pamięci czyli liczba bloków, błędnie nazwany rozmiarem bloku, wynosi 256. Liczba skrótów na sekundę to $63,56MHz/64\frac{cykle}{Hash} \approx 995kHash/s$. Gdyby przepustowość zdefiniować dla danych na wyjściu układu, to dla 128-bitowego skrótu by wynosiła 121,4Mbit/s, a dla 256-bitowego 242,8Mbit/s.

Na listach mailingowych związanych z konkursem PHC pojawiały się w 2015 roku wzmianki dotyczące wyników uzyskanych przez Bielec [18] [19] [20] na różnych GPU zaprogramowanych w języku OpenCL. W sieci nie ma żadnych innych dalszych śladów po oryginalnej implementacji, natomiast są ogólnodostępne prawdopodobne klony re-

pozytorium [21]. Nie znaleziono żadnych informacji o próbach wykorzystania narzędzi różnych producentów umożliwiających automatyczną implementację algorytmów na podstawie kodu OpenCL, na układy FPGA do realizacji tego algorytmu.

W 2017 roku Rossetti [22] zaprezentował implementację funkcji oznaczonej jako P na rys.2.2 dla układu ASIC wykonanego w technologii 90nm. Porównanie jej z proponowaną w pracy implementacją dla układu SoC FPGA przedstawiono w rozdziale 6.4.

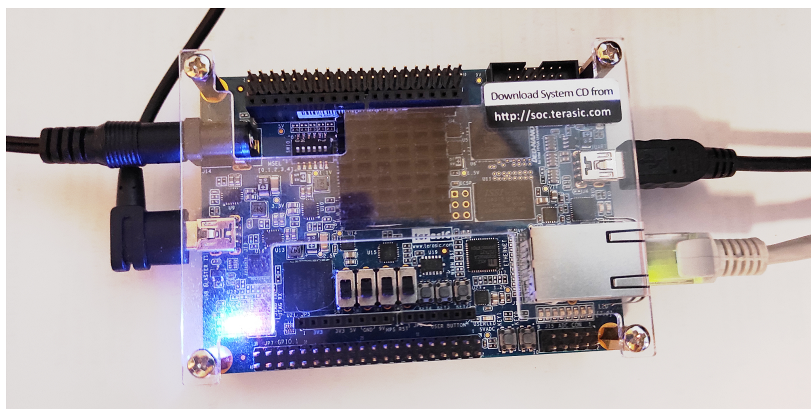
Atiwa [23] w 2020 roku przedstawił koprocesor FPGA realizujący sprzętowo algorytm Blake2b. Porównał również implementację dla układu SoC FPGA z implementacją gdzie koprocesor w FPGA został podłączony do procesora w zwykłym komputerze poprzez PCIe. Implementacja na SoC co prawda osiągnęła 5-krotnie mniejszą liczbę skrótów na sekundę, ale za to pobierała prawie 15 razy mniej mocy, co się przekłada na praktycznie 3-krotny spadek mocy w przeliczeniu na skrót.

LYrA2 jest jednym z wyróżnionych algorytmów w konkursie PHC [6]. Implementację na układ FPGA opublikował Van Beirendonck [24] w 2019. Tak jak Argon2, był projektowany z myślą o utrudnianiu prowadzenia efektywnych obliczeń na GPU czy układach ASIC. Również bazuje na funkcji P z Blake2b. Jednak w związku z tym, że autorzy przedstawili jedynie sam rdzeń realizujący algorytm, a nie cały system oraz była to pierwsza implementacja tego algorytmu na układ FPGA, nie opublikowali wyników żadnych porównań.

3. Koncepcja systemu

3.1. Założenia

Realizowany system miał zostać uruchomiony na płycie *Terasic DE0-Nano-SoC*, zawierającej układ *SoC* z rodziny *Intel Cyclone V*. Na podstawie wyników profilowania oraz analizy algorytmu należało zaprojektować i opisać w języku VHDL koprocesor realizujący wybrany fragment algorytmu, i zaimplementować go w części FPGA dostępnej w ramach *SoC*. Należało również zapewnić mechanizmy umożliwiające uruchamianym na *HPS* aplikacjom wymianę danych z koprocesorem oraz sterowanie nim. Mianem ***HPS*** *(Hard Processor System)* producent określa procesor *ARM Cortex-A9* wraz z peryferiami w niereprogramowalnej części czipu.



Rysunek 3.1. Płyta *Terasic DE0-Nano-SoC*

3.2. Profilowanie kodu referencyjnego

W celu analizy czasu wykonywania poszczególnych części algorytmu przygotowano referencyjną implementację algorytmu do profilowania [15]. Wybrano implementację w standardzie *C99*, wersję bez dodatkowych optymalizacji, ponieważ na płycie dostępny jest procesor o architekturze *ARMv7*, a nie *x86*.

Zmodyfikowano *Makefile* programu oraz stworzono pakiet umożliwiający zainstalowanie programu.

Do profilowania wykorzystano program *Gprof*. Program w regularnych odstępach, najczęściej co 10ms, generuje przerwanie i sprawdza jakie funkcje są obecne w stosie wywołań. W związku z powyższym, otrzymane wyniki stanowią tylko statystyczne przybliżenie, wnosząc pewne błędy do pomiaru dla krócej wykonywanych funkcji. Dodatkowo, żeby zredukować wpływ błędów dla krótko wykonywanych funkcji, w analizowanym programie po sparsowaniu argumentów i otwarciu urządzenia wywoływano funkcję realizującą algorytm w pętli. Dokonano 12 pomiarów zmieniając liczbę powtórzeń pętli (100, 1000 oraz 10000) oraz parametr `tcost` algorytmu (1,2,3,5,7 oraz 10) odpowiadający liczbie iteracji algorytmu wykonywanych nad każdym blokiem danych.

Tabela 3.1. Profilowanie algorytmu na procesorze ARM, parametr `tcost`: 1, liczba powtórzeń: 10000.

Nazwa	% Czasu	Średni czas wykonywania [ns]	Liczba wywołań	Czas [s]
FillBlock	58,70	26681	2320000	61,90
blake2b_compress	29,69	12089	2590000	31,31
XORBlock	8,22	1857	4670000	8,67
__udivmoddi4	1,16	-	-	1,22
blake2b_final	0,43	180	2500000	0,45
blake2b_init_param	0,37	156	2500000	0,39
FillSegment	0,29	1938	160000	0,31
blake2b_long	0,18	2111	90000	0,19
blake2b	0,17	75	2400000	0,18
IndexAlpha	0,16	73	2320000	0,17
blake2b_init	0,15	64	2500000	0,16
CopyBlock	0,10	16	6970000	0,11
blake2b_update	0,10	40	2720000	0,11
__aeabi_uidiv	0,09	-	-	0,09
__aeabi_uldivmod	0,07	-	-	0,07
Argon2Core	0,02	2000	10000	0,02
Argon2d	0,02	2000	10000	0,02
FillMemoryBlocks	0,02	2000	10000	0,02
FillSegmentThr	0,02	-	-	0,02
__aeabi_uidivmod	0,02	-	-	0,02
FillFirstBlocks	0,01	1000	10000	0,01
Run	0,01	1000	10000	0,01
secure_wipe_memory	0,00	0	30000	0,00
AllocateMemory	0,00	0	10000	0,00
ClearMemory	0,00	0	10000	0,00
Finalize	0,00	0	10000	0,00
FreeMemory	0,00	0	10000	0,00
InitialHash	0,00	0	10000	0,00
Initialize	0,00	0	10000	0,00
ValidateInputs	0,00	0	10000	0,00

W tab.3.1 przedstawiono wyniki dla pomiaru z liczbą powtórzeń wynoszącą 10000 oraz parametrem `tcost` równym 1. Ponieważ blok danych jest mieszany tylko raz, to spośród wszystkich pomiarów dla różnej wartości parametru *tcost*, to właśnie w tym pomiarze funkcje inicjalizujące i finalizujące przetwarzanie bloku danych będą miały najwyższy procentowy udział w całkowitym czasie wykonywania algorytmu. Najwięcej można zyskać akcelerując funkcję `FillBlock`. Drugi w kolejności możliwy jest zysk wynikający z akceleracji `blake2b_compress`. Wymierne, acz znacznie mniejsze w skali czasu liczenia całości

w porównaniu z poprzednimi, przyspiesznie można spróbować uzyskać przyspieszając operację `XORBlock`, która xoruje ze sobą 2 bloki, a jej wynik nadpisuje pierwszy argument.

Na podstawie powyższych wyników zdecydowano, że przy pomocy części programowalnej układu przyspieszana będzie funkcja `FillBlock`. Po analizie teoretycznej algorytmu stwierdzono, że koprocesor będzie realizował fragment kodu referencyjnego przedstawiony na listingu 3.1, wyróżniony dodatkowo kolorem ceglastym na rys.2.2. W HPS pozostanie realizacja fazy inicjalizacji i finalizacji algorytmu. Dodatkowo, żeby dwukrotnie obniżyć ilość wymienianych danych między HPS a koprocesorem, operacja xorowania bloków przed, jak i po mieszaniu, będzie realizowana przez HPS.

Jeśli wyniki akceleracji okażą się zadowalające, warte rozważenia będzie również akcelerowanie funkcji `blake2b_compress`. Funkcja `FillBlock` opiera się w dużej mierze na zmodyfikowanej `blake2b_compress`, wykonywanej dwukrotnie. Umożliwia to potencjalnie współdzielenie większości zasobów sprzętowych koprocesora przez obie funkcje, ograniczając wzrost zużycia zasobów do minimum. Funkcja `XORBlock` jest z kolei dobrym kandydatem do akceleracji przy wykorzystaniu rozszerzenia *NEON* architektury *ARM*. Można by spróbować dzięki temu przyspieszyć liczenie wyniku xorowania 2 bloków poprzez zastosowanie odpowiedniej instrukcji SIMD *(Single Instruction Multiple Data)* [25].

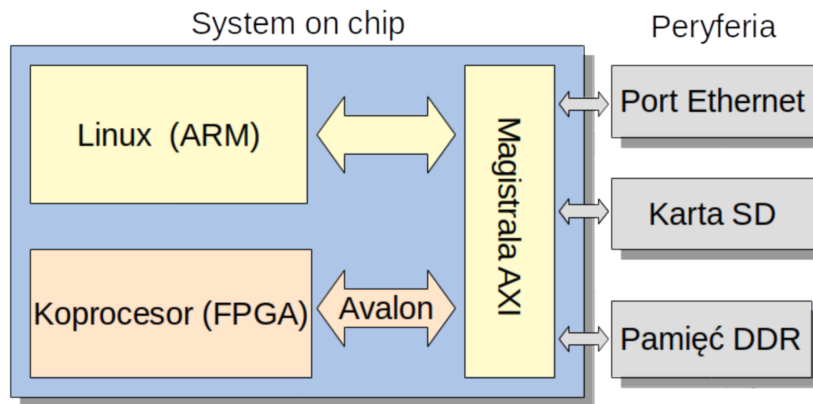
```c
#include<stdint.h>
#define rotr64(x, y)  (((x) >> (y)) ^ ((x) << (64 - (y))))
#define G(a,b,c,d)          \
        a = fBlaMka(a, b) ;    \
        d = rotr64(d ^ a, 32); \
        c = fBlaMka(c, d);     \
        b = rotr64(b ^ c, 24); \
        a = fBlaMka(a, b) ;    \
        d = rotr64(d ^ a, 16); \
        c = fBlaMka(c, d);     \
        b = rotr64(b ^ c, 63);
#define BLAKE2_ROUND_NOMSG(v0,v1,v2,v3,v4,v5,v6,v7,v8,v9,v10,v11,v12,v13,v14,v15)  \
        G(v0, v4, v8,  v12);  \
        G(v1, v5, v9,  v13);  \
        G(v2, v6, v10, v14);  \
        G(v3, v7, v11, v15);  \
        G(v0, v5, v10, v15);  \
        G(v1, v6, v11, v12);  \
        G(v2, v7, v8,  v13);  \
        G(v3, v4, v9,  v14);
/*designed by the Lyra PHC team */
static inline uint64_t fBlaMka(uint64_t x, uint64_t y)
{
        uint32_t lessX = (uint32_t)x;
        uint32_t lessY = (uint32_t)y;
        uint64_t lessZ = (uint64_t)lessX;
        lessZ = lessZ * lessY;
        lessZ = lessZ << 1;
        uint64_t z = lessZ + x + y;
        return z;
}
int argon2d_fpga(uint64_t *block)
{
  // Apply Blake2 on columns of 64-bit words: (0,1,...,15) ,
  // then (16,17,..31)... finally (112,113,...127)
  for (unsigned i = 0; i < 8; ++i) {
    BLAKE2_ROUND_NOMSG(block[16*i],    block[16*i+1],  block[16*i+2],  block[16*i+3],
                       block[16*i+4],  block[16*i+5],  block[16*i+6],  block[16*i+7],
                       block[16*i+8],  block[16*i+9],  block[16*i+10], block[16*i+11],
                       block[16*i+12], block[16*i+13], block[16*i+14], block[16*i+15]);
   }
  // Apply Blake2 on rows of 64-bit words: (0,1,16,17,...112,113),
  // then (2,3,18,19,...,114,115).. finally (14,15,30,31,...,126,127)
  for (unsigned i = 0; i < 8; i++) {
    BLAKE2_ROUND_NOMSG(block[2*i],    block[2*i+1],  block[2*i+16],  block[2*i+17],
                       block[2*i+32], block[2*i+33], block[2*i+48],  block[2*i+49],
                       block[2*i+64], block[2*i+65], block[2*i+80],  block[2*i+81],
                       block[2*i+96], block[2*i+97], block[2*i+112], block[2*i+113]);
  }
  return 0;
}
```

Listing 3.1. Fragment kodu w C realizowany przez koprocesor

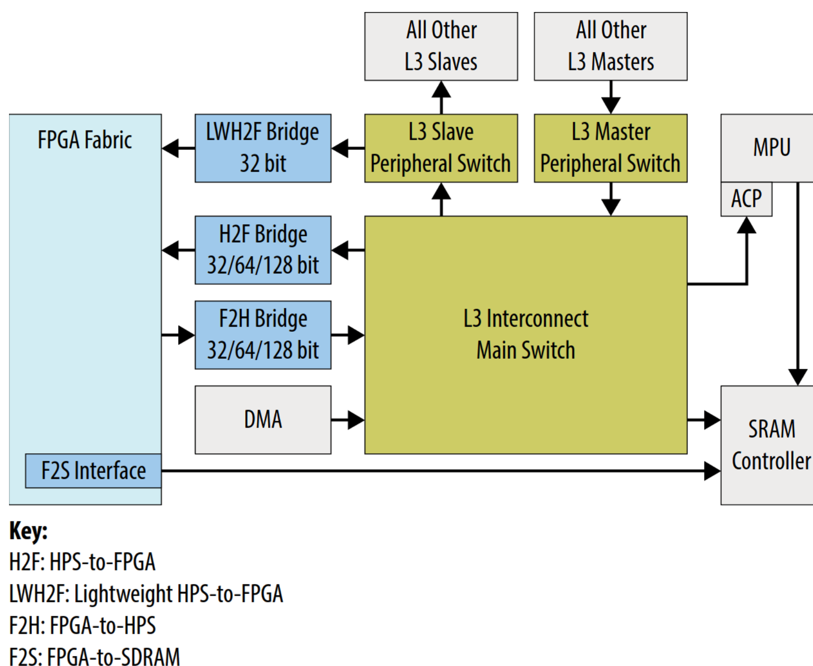
3.3. Ogólna struktura systemu

3.3.1. Część sprzętowa

Na rys.3.2 schematycznie przedstawiono architekturę systemu.



Rysunek 3.2. Ogólna struktura systemu



Rysunek 3.3. Schemat blokowy ilustrujący architekturę układu *Cyclone V System-on-Chip* [26].

Po uruchomieniu systemu, zmodyfikowana procedura inicjalizacji będzie ładować moduł jądra systemu ze sterownikiem do koprocesora. Sterownik podczas ładowania zaalokuje w pamięci RAM DDR ciągły obszar, służący jako bufor wymiany danych pomiędzy HPS, a koprocesorem. Wymiana danych będzie następować przy wykorzystaniu techniki DMA *Direct Memory Access*, stąd dalej w pracy ten bufor będzie nazywany **buforem DMA**. W obrębie HPS bloki funkcjonalne są połączone przy pomocy magistrali *AXI*.

Żeby podłączyć koprocesor do reszty systemu, zostanie stworzone połączenie pomiędzy koprocesorem a tzw. *Interconnectem* przy pomocy magistrali *Avalon-MM*.

Dostępny na płycie port Ethernet umożliwi podpięcie płyty do sieci oraz zdalny dostęp do systemu przez SSH.

3.3.2. Część programowa

Przełączniki dostępne na płycie ustawiono tak, aby system był uruchamiany (*bootowany*) z karty SD. Po podłączeniu zasilania do płyty, będzie zaczynała się procedura bootowania [27]. W pierwszej kolejności procesor wczyta z pamięci ROM procedurę załadowania tzw. *preloadera*, nazywanego inaczej *First Stage Bootloader, FSBL*. W roli preloadera wykorzystano *U-Boot SPL*. Jest to okrojona wersja *U-Boot*, która będzie inicjalizować niezbędne minimum zasobów sprzętowych, w celu umożliwienia załadowania pełnej wersji *U-Boot (Second Stage Bootloader)* na procesor. *U-Boot* po uruchomieniu będzie inicjalizował resztę zasobów, programował układ FPGA oraz uruchamiał system operacyjny. W przypadku połączenia z płytą przez interfejs szeregowy, możliwe jest na tym etapie przerwanie automatycznej procedury startu i ręczna praca z konsolą *U-Boota*. Umożliwia to edycję ustawień środowiska *U-Boota* już po wygenerowaniu obrazów na karcie SD oraz w przypadku wgrania wadliwej aktualizacji obrazów systemu, uruchomienie zapasowej kopii z minimalnymi, działającymi obrazami.

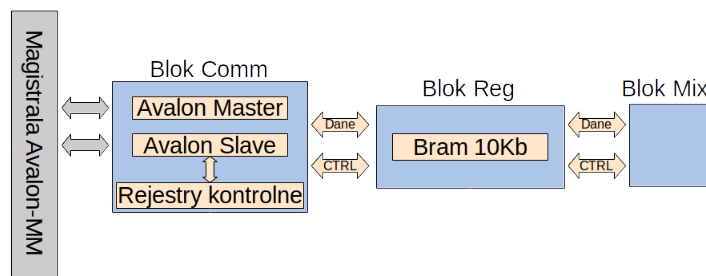
Na procesorze *ARM* uruchomiony zostanie system operacyjny Linux. Następnie zostanie załadowany sterownik koprocesora, a urządzenie zostanie otwarte. Od razu po uruchomieniu, aplikacja użytkownika skonfiguruje koprocesor wykorzystując komendy *ioctl* udostępnione przez sterownik, a potem przejdzie do prowadzenia obliczeń. W czasie trwania obliczeń HPS z koprocesorem wymieniają między sobą pojedyncze bloki danych o rozmiarze 1KB,

3.4. Architektura koprocesora

Z powodu braku możliwości przeniesienia całości algorytmu do części programowalnej układu *SoC FPGA* [28], niemożliwym jest zastosowanie przetwarzania potokowego czy podobnych, zaawansowanych metod optymalizacji realizacji sprzętowych. Dla akceleracji tylko wybranych części algorytmu, kluczowe jest zapewnienie minimalnej latencji związanej z przetwarzaniem. W takiej sytuacji, to przepustowość łącza pomiędzy koprocesorem a pamięcią RAM będzie najprawdopodobniej stanowić główne ograniczenie. W związku z tym, architektura koprocesora w maksymalnym stopniu powinna umożliwiać równoczesny transfer oraz przetwarzanie danych. Na tej podstawie założono, że operacja mieszania danych będzie jak najkrótsza oraz będzie trwać nie dłużej niż transfer wiersza bloku, tj. 32 takty zegara.

Na rys.3.4 przedstawiono proponowaną architekturę koprocesora. Całość podzielono na 3 bloki funkcjonalne:

1. **blok Comm** - odpowiadający za konfigurację koprocesora oraz komunikację z resztą systemu poprzez magistralę Avalon-MM,
2. **blok Reg** - przechowujący blok danych w pamięci BRAM oraz kontrolujący proces mieszania kolejnych fragmentów bloku,
3. **blok Mix** - realizujący funkcję mieszania.



Rysunek 3.4. Ogólna architektura koprocesora

W bloku *Comm* zatem można wyróżnić 2 podbloki. Pierwszy z nich musi nasłuchiwać poleceń napływających ze strony HPS oraz przechowywać zestaw danych konfigurujących koprocesor. Dodatkowo powinien umożliwiać wykonanie tzw. miękkiego resetu koprocesora, który polega na zresetowaniu pozostałych bloków koprocesora, jednocześnie zachowując dane konfigurujące. Drugi natomiast powinien być w stanie zlecić transakcję odczytu danych z bufora DMA w trybie *burst* i przekazywać te dane na bieżąco dalej, do bloku *Reg*. Żeby umożliwić poprawne dokonywanie transakcji zapisu danych do pamięci DDR, blok musi posiadać wewnętrzny bufor, przechowujący ostatnio wysyłane słowo, umożliwiający jego ponowne nadanie w przypadku otrzymania żądania retransmisji.

Blok *Reg* będzie odpowiadał za obsługę pamięci BRAM. Powinien po każdym przesłanym wierszu, czyli 1/8 bloku, umożliwiać przekazywanie danych do mieszania, zanim cały blok zostanie odczytany przez koprocesor. Ponieważ przerwy w transmisji danych napływających od strony bloku *Comm* mogą występować w sposób niedeterministyczny, niemożliwym jest w sposób synchroniczny zaplanowanie z góry operacji odczytów/zapisów między blokami *Comm* i *Mix*. Zatem *Reg* musi zawierać jakiś mechanizm arbitrażu przydzielający dostęp do pamięci BRAM w danym momencie tylko i wyłącznie jednemu blokowi.

Blok *Mix* powinien być w stanie w jak najkrótszym czasie odczytać dane z bloku *Reg*, wyznaczyć wynik funkcji mieszania dla odczytanych danych, a następnie zapisać otrzymane dane z powrotem w *Reg*.

3.5. Dostosowanie części programowej do współpracy ze sprzętem

Żeby umożliwić wykorzystanie funkcji realizowanych sprzętowo w części programowej systemu, koniecznie będzie przygotowanie sterownika do koprocesora. Będzie on musiał implementować zbiór funkcji umożliwiających zapisy i odczyty z rejestrów urządzenia

oraz przygotowujących w pamięci RAM bufor DMA. Przygotowany sterownik będzie wkompilowany w jądro systemu jako moduł. Aplikacje będą mogły korzystać z funkcji udostępnianych przez sterownik przy wykorzystaniu zestawu komend *ioctl*.

Następnie będzie trzeba zmodyfikować kod źródłowy aplikacji. Żeby skorzystać z koprocesora aplikacja będzie musiała w pierwszej kolejności zainicjalizować koprocesor, a następnie pobrać adres bufora DMA. Koniecznym będzie rozszerzenie różnych struktur czy zestawów argumentów funkcji w celu umożliwienia przekazywania wskaźnika na bufor DMA. Przeniesione w sprzęt fragmenty funkcji będzie należało zastąpić kolejno:

- zapisem danych do bufora,
- uruchomieniem koprocesora,
- aktywnym oczekiwaniem na zakończenie pracy koprocesora,
- odczytem danych z bufora,
- zatrzymaniem koprocesora.

Żeby wykorzystać rozszerzenie *Neon* [29], będzie trzeba zastąpić przyszpieszane funkcje wywołaniami odpowiednich instrukcji SIMD.

Na koniec, mając już gotowy kod zarówno sterownika jak i mierzonych aplikacji, konieczne będzie przygotowanie obrazu systemu, który można będzie uruchomić na płycie. W tym celu potrzebne będzie stworzenie pakietu do narzędzia *Buildroot* ze sterownikiem urządzenie, a następnie dodanie go do listy budowanych pakietów. Dodatkowo przełączenia będzie wymagała opcja dodająca obsługę rozszerzenia *Neon*.

4. Implementacja

4.1. Implementacja koprocesora

Wszystkie bloki opisano w języku VHDL zgodnie z zaleceniami Gaislera [30], dzieląc je w miarę możliwości na 2 procesy: sekwencyjny i kombinacyjny. Pewien wyjątek stanowi blok *Comm*, gdzie dodatkowy proces odpowiada za implementację interfejsu *Avalon Slave* i przechowywanie konfiguracji koprocesora. W procesie kombinacyjnym wyznaczany jest następny stan wszystkich rejestrów, zorganizowanych w kodzie jako rekord. Natomiast w procesie sekwencyjnym następuje albo synchroniczny reset, równoważny przypisaniu rekordowi wartości jaką jest inicjalizowany, albo przepisanie wartości wyznaczonych jako następne.

```
am_seq : process (clock_sink_clk) is
begin  -- process am_seq
if rising_edge(clock_sink_clk) then
  if ((reset_sink_reset = '1') or (ctrl(1) = '1')) then  -- synchronous reset
  ↪  (active high)
    am_r <= AM_REGS_INIT;
  else
    am_r <= am_r_n;
  end if;

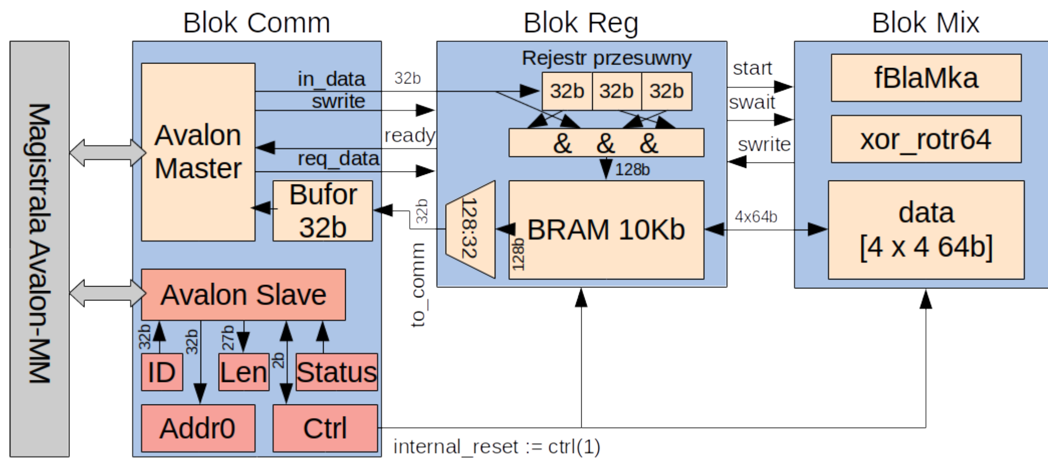
end if;
end process am_seq;
```

Listing 4.1. Przykładowy proces sekwencyjny odpowiadający za przypisanie wartości do rekordu `r`.

Taka konwencja opisu sprzętu w większości przypadków podnosi czytelność kodu. Upraszcza inicjalizowanie/resetowanie wartości w procesach, ponieważ zamiast wielu pojedynczych przypisań w wielu liniach dla poszczególnych rejestrów, można zastosować pojedyncze przypisanie dla całego rekordu. W analogiczny sposób można stworzyć rekord sygnałów kombinacyjnych i je wspólnie inicjalizować, czy też rekord zmiennych, jeśli w procesie wykorzystywana jest ich większa ilość. Kolejną znaczącą zaletą takiego podejścia jest możliwość uniknięcia powstawania w kodzie przypadkowych zatrzasków. Narzuca ono następującą strukturę kodu:

- inicjalizacja rekordu nowych wartości rejestrów `r_next`, rekordu sygnałów kombinacyjnych `c` oraz ewentualnego rekordu zmiennych `v` następuje na samym początku procesu kombinacyjnego;
- rekordy `r_next` oraz `c` w procesie kombinacyjnym mogą występować tylko z lewej strony operatora przypisania, rekord rejestrów `r` tylko z prawej;
- zakładając że reset jest synchroniczny, w procesie sekwencyjnym na narastającym zboczu zegara następuje przypisanie wartości początkowej do rekordu `r` jeśli wystąpił reset, w przeciwnym przypadku przepisanie rekordu `r_next` do rekordu `r`.

Wykorzystując dowolny edytor kodu wspierający składnię VHDL, z opcją automatycznego generowania listy czułości procesów, np. *Emacs*, można w prosty sposób zweryfikować czy kod został napisany w sposób zgodny z powyższymi ograniczeniami. W poprawnie napisanym kodzie, automatycznie wygenerowana lista czułości nie może zawierać rekordów `r_next` oraz `c`.



Rysunek 4.1. Proponowana struktura koprocesora. Kolorem ceglastym zaznaczono część odpowiadającą za konfigurację koprocesora.

Tabela 4.1. Porty koprocesora.

Nazwa	Kierunek	Typ
`clk`	`in`	`std_logic`
`rst`	`in`	`std_logic`
`Avalon Master`		
`avalon_master_address`	`out`	`std_logic_vector(31 downto 0)`
`avalon_master_burstcount`	`out`	`std_logic_vector(27 downto 0)`
`avalon_master_waitrequest`	`in`	`std_logic`
`avalon_master_response`	`in`	`std_logic_vector(1 downto 0)`
`avalon_master_write`	`out`	`std_logic`
`avalon_master_writedata`	`out`	`std_logic_vector(31 downto 0)`
`avalon_master_read`	`in`	`std_logic`
`avalon_master_readdata`	`in`	`std_logic_vector(31 downto 0)`
`avalon_master_readdatavalid`	`in`	`std_logic`
`Avalon Slave`		
`avalon_slave_address`	`in`	`std_logic_vector(2 downto 0)`
`avalon_slave_waitrequest`	`out`	`std_logic`
`avalon_slave_response`	`out`	`std_logic_vector(1 downto 0)`
`avalon_slave_read`	`in`	`std_logic`
`avalon_slave_readdata`	`out`	`std_logic_vector(31 downto 0)`
`avalon_slave_write`	`in`	`std_logic`
`avalon_slave_writedata`	`in`	`std_logic_vector(31 downto 0)`

Całość zawarto w pojedynczym pliku .vhd, w którym zainstancjowano i połączono ze sobą wszystkie bloki. Porty koprocesora przedstawiono w tabeli 4.1.

4.2. Blok Comm

Blok Comm zawiera parę interfejsów *Avalon Master* oraz *Avalon Slave*. Zaimplementowano je zgodnie z oficjalną specyfikacją [31]. Interfejs *Master* dokonuje transakcji w trybie *burst*, *Slave* - pojedynczych zapisów lub odczytów. Zrezygnowano z implementacji większości opcjonalnych sygnałów ograniczając ich zestaw do koniecznego minimum.

```
avalon_slave_waitrequest <= '1' when ((avalon_slave_read = '1') and (as_read_ack
↪ = '0')) or (reset_sink_reset = '1') else '0';
as : process (clk) is
begin  -- process as
if rising_edge(clk) then
  avalon_slave_readdata <= (others => '0');
  avalon_slave_response <= (others => '0');
  as_read_ack           <= '0';
  if avalon_slave_read = '1' then
    as_read_ack <= '1';
    case addr is
      when 0 =>
        avalon_slave_readdata <= id;
      ...
      when others =>
        avalon_slave_response <= "11";
    end case;
  end if;
  if avalon_slave_write = '1' then
    case addr is
      when 1 =>
        ctrl <= avalon_slave_writedata(1 downto 0);
      ...
      when others =>
        avalon_slave_response <= "11";
    end case;
  end if;
  if ctrl(1) = '1' then
    ctrl(1) <= '0';
  end if;
end if;
end process as;
```

Listing 4.2. Proces implementujący interfejs *Avalon Slave*.

Slave umożliwia konfigurację oraz monitorowanie stanu koprocesora. Jest to jedyna część koprocesora opisana pojedynczym procesem zamiast pary. W procesie synchronicznym następuje w następnym takcie zegara ustawienie `as_read_ack` służącego do

asynchronicznego generowania odpowiedzi *waitrequest*, odczyt lub zapis do wskazanego w żądaniu rejestru oraz wyzerowanie bitu rejestru `ctrl(1)`, jeśli ten jest podniesiony.

Tabela 4.2 przedstawia układ rejestrów, wraz z opisem ich funkcji. Ponieważ szyna danych w magistrali ma 32 bity szerokości, bity nieujęte w tabeli są ignorowane i mogą przyjmować dowolne wartości.

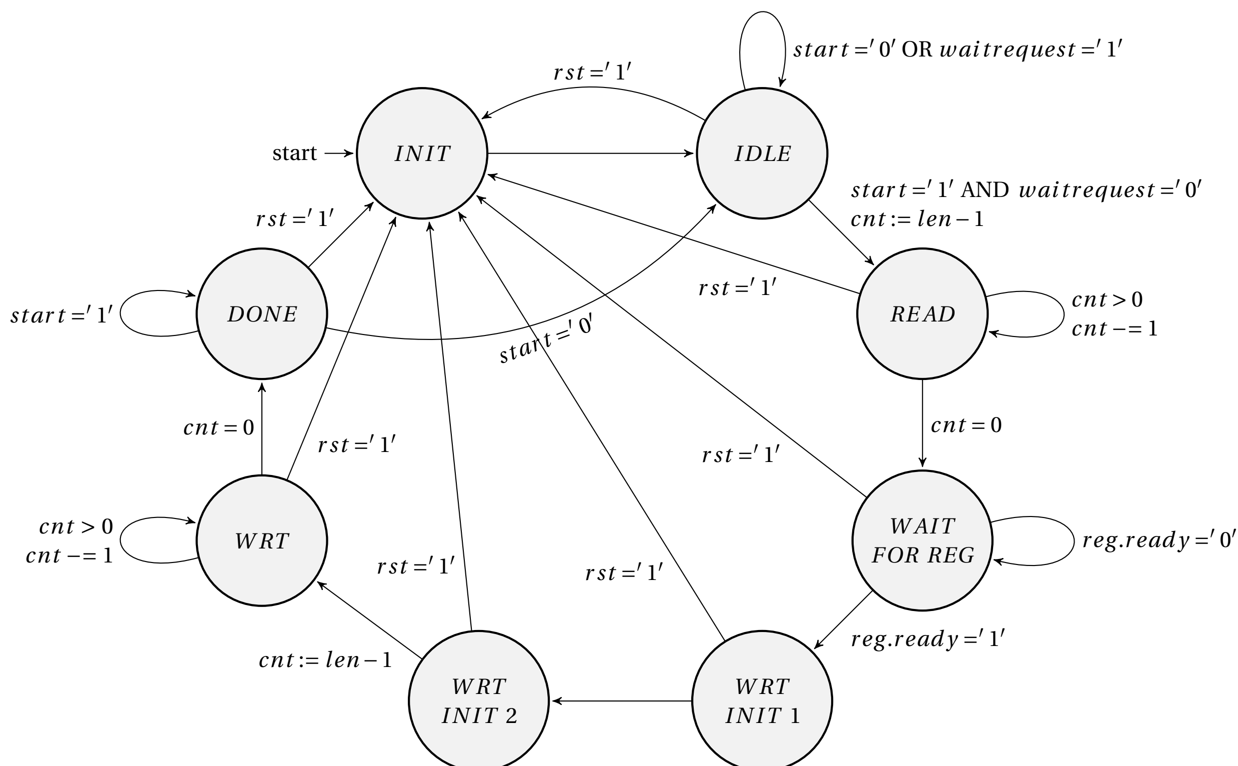
Tabela 4.2. Rejestry koprocesora.

Adres	Bity	Nazwa	Możliwość odczytu (R) / zapisu (W)	Opis
0	31:0	ID	R	ID koprocesora, umożliwia sterownikowi urządzenia kontrolę wersji.
1	1:0	Ctrl	R+W	Rejestr kontrolny. Bit pierwszy ustawiony na 1 umożliwia miękki reset koprocesora. Po restarcie zakończonym sukcesem, automatycznie zmienia wartość na 0. Bit zerowy ustawiony na 1 uruchamia koprocesor, przełączony z 1 na 0 powoduje reset koprocesora po zakończeniu obliczeń. Tak wywołany reset nie zeruje zawartości rejestrów opisanych w tej tabeli.
2	0	Status	R	Przechowuje wartość 1 kiedy koprocesor skończy zapis danych wynikowych pod zaprogramowanym adresem. W przeciwnym przypadku przechowuje 0.
3	31:0	Addr0	W	Adres fizyczny początku bufora DMA.
4	27:0	Len	W	Długość bufora, dla Argon2d należy zawsze ustawiać na 256

Interfejs *Master*, po wcześniejszym skonfigurowaniu rejestrów koprocesora, służy do odczytu danych wejściowych oraz zapisu danych wynikowych. Graf stanów przedstawiono na rys.4.2.

W stanie *IDLE* oczekuje na polecenie startu, tzn. na moment kiedy najmłodszy bit rejestru *Ctrl* zostanie przestawiony z ′0′ na ′1′. Następnie rozpoczyna transakcję odczytu w trybie *Burst* z pamięci DDR. Polega ono na:

- wystawieniu długości transakcji na sygnale *burstcount*,
- podaniu adresu początkowego na sygnale *address*,
- przypisaniu ′1′ do sygnału *read*.



Rysunek 4.2. Uproszczony graf stanów bloku *Comm*

Po rozpoczęciu transakcji, ze strony magistrali powinna napłynąć odpowiedź. Jeśli sygnał *waitrequest* ma wartość $'1'$, to układ nie zmieniając stanu wyjść oczekuje na zmianę tej wartości. Jeśli natomiast ma wartość $'0'$, to w następnym takcie układ przechodzi do następnego stanu oraz inicjalizuje licznik słów wartością z rejestru *len* zmniejszoną o 1.

W stanie *READ* następuje odczyt napływających z magistrali słów. W każdym takcie kiedy jest odbierane słowo, czyli sygnał *readdatavalid* jest równy $'1'$, wartość licznika słów jest zmniejszana o 1, a odczytane słowo jest bezpośrednio przekazywane do bloku *Reg*. Takt po tym jak wartość licznika słów wyniesie 0, układ przejdzie do stanu oczekiwania na zakończenie pracy przez dalsze bloki oraz ponownie zainicjalizuje licznik słów wartością z rejestru *len* zmniejszoną o 1.

Po otrzymaniu informacji z bloku *Reg*, że wyniki są gotowe do odesłania do bufora DMA, *Comm* rozpoczyna procedurę zapisu. W pierwszej kolejności wysyła do *Reg* żądanie przygotowania danej, a następnie przechodzi do stanu *WRT_INIT_1*. Otrzymane od *Reg* słowo zapisuje w wewnętrznym buforze umożliwiającym retransmisję ostatniego słowa w razie potrzeby, wysyła kolejne żądanie przygotowania następnego słowa i przechodzi do stanu *WRT_INIT_2*. Rozpoczyna transakcję zapisu w trybie *Burst* poprzez:

- podanie adresu początkowego,
- określenie długości transakcji,
- przypisaniu $'1'$ do sygnału *write*,
- przypisanie słowa z bufora do sygnału *writedata*.

Jeśli odpowiedź *waitrequest* jest równa ′1′, to układ oczekuje, aż ta się zmieni. W przeciwnym razie, zmniejsza wartość licznika o 1, zapisuje do bufora słowo wystawione przez *Reg*, żąda przygotowania następnego słowa i przechodzi do stanu *WRT*.

W stanie *WRT*, dopóki licznik słów ma niezerową wartość, *Comm* ustawia sygnał *write* w stan wysoki oraz przepisuje słowo z bufora na wyjście *writedata*. W przypadku wystąpienia przerwy w transmisji, wartości wyjść zostają podtrzymane. Natomiast gdy *waitrequest* ma wartość ′0′, to zmniejszana jest wartość licznika o 1 oraz na wyjście przepisywane jest słowo z bufora. Jeśli licznik ma wartość nie mniejszą niż 2 to dodatkowo przekazywane do *Reg* jest żądanie przygotowania kolejnego słowa.

Gdy licznik osiągnie wartość 0 oraz *waitrequest* jest w stanie niskim, to oznacza że ostatnie słowo zostaje wysłane, a blok *Comm* przechodzi do stanu *DONE*, gdzie oczekuje na twardy, albo miękki reset. Twardy reset powodowany jest poprzez podniesienie globalnego sygnału reset w części programowalnej i powoduje całkowity reset koprocesora, łącznie z rejestrami kontrolnymi.

4.3. Blok Reg

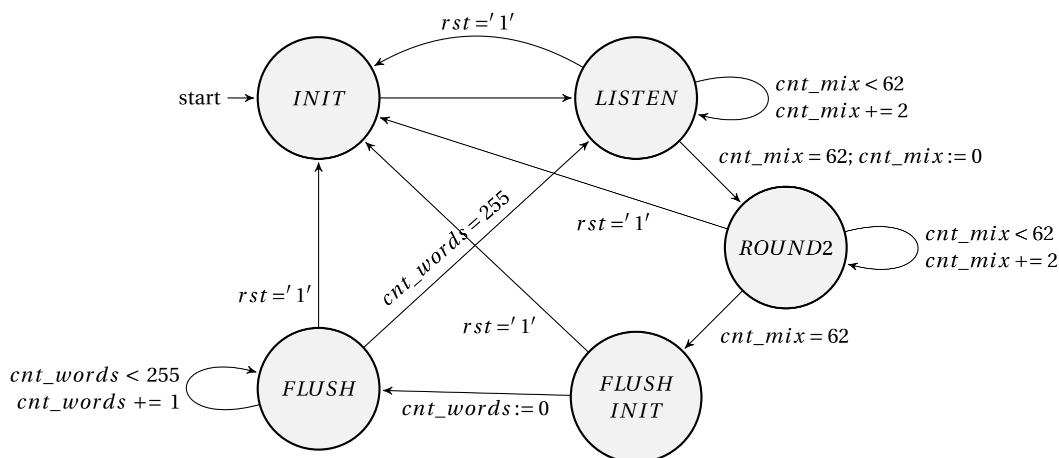
Blok Reg realizuje pamięć podręczną, wraz z jej kontrolerem. Zawiera w sobie rdzeń *IPCore* od firmy Intel, implementujący 2-portowy RAM [32], który wykorzystuje pojedynczą komórkę pamięci *BRAM M10K*. Blok danych jest w pamięci reprezentowany jako macierz 8x8 słów 128-bitowych, adresowanych kolejno wierszami. Ponieważ z magistrali napływają słowa o szerokości 32 bitów, to na wejściu bloku umieszczono rejestr przesuwny o długości 3 słów. Umożliwia to zszywanie danych z bufora, wraz z kolejnym słowem z wejścia, w słowa o szerokości 128 bitów, które następnie zostają zapisane w pamięci. Dane od razu są zszywane z wymaganym przez algorytm przeplotem.

W koprocesorze liczone są wyniki 2 rund mieszania, zatem blok *Reg* posiada 3 główne stany:

- *LISTEN*, w którym oczekuje na przyjście danych oraz kontroluje 1. rundę w *Mix*,
- *ROUND2*, w którym odpowiada za przeprowadzenie 2. rundy mieszania,
- *FLUSH*, w którym odczytuje po kolei wyniki z pamięci BRAM i przekazuje je do bloku *Comm*.

Dodatkowo posiada 2 stany pomocnicze. Stan *INIT* to stan do którego trafia na skutek resetu. Następuje w nim załadowanie domyślnych wartości do rejestrów bloku. Stan *FLUSH_INIT* natomiast, to stan w którym blok sygnalizuje gotowość do rozpoczęcia zwracania danych, poprzez ustawienie wyjścia *data_ready* w stan wysoki oraz w którym zerowane są liczniki słów oraz plastrów, czyli kolejnych 32-bitowych fragmentów słowa. Schematycznie przedstawiono automat na rys.4.3.

W celu redukcji czasu prowadzenia obliczeń przez koprocesor, *Reg* po zapisaniu każdego pełnego wiersza (co 32 słowa o długości 32 bitów) uruchamia od razu procedurę mieszania wiersza. Po ustawieniu bitu startu dla bloku *Mix* w stan wysoki, *Reg* przez 4



Rysunek 4.3. Uproszczony graf stanów bloku *Reg*

kolejne takty zegara odczytuje z pamięci BRAM po 2 słowa 128b i przesyła je do bloku *Mix*. Po pewnym czasie *Mix* zwraca przez 4 kolejne takty wyniki mieszania. Ponieważ równocześnie *Comm* przekazuje na bieżąco dane z magistrali, to napływające słowa trzeba zapisywać co minimum 4 takty.

Potencjalnie prowadzi to do konfliktu, ponieważ każdy port pamięci może jednocześnie tylko albo odczytywać, albo zapisywać tylko i wyłącznie jedno słowo. W związku z tym przyjęto, że priorytet ma zapis napływających z bloku *Comm* danych, a w czasie takiego zapisu *Mix* jest wstrzymywany na 1 takt zegara. Za każdym razem, gdy wartość licznika plastrów wynosi 3, czyli w takcie, w którym zlecany będzie zapis słowa wejściowego z bufora, do pamięci BRAM, podnoszona jest wewnętrzna flaga *wait_flag*. W przypadku kiedy możliwy jest zapis do pamięci BRAM przez *Mix*, czyli kiedy nie trwa odczyt danych przekazywanych do *Mix*, a flaga jest w górze, sygnał *swait* jest ustawiany na wartość ′1′. Kiedy trwa przekazywanie danych z BRAM do *Mix*, wartość licznika *cnt_mix_trigger* jest różna od zera. Wtedy gdy flaga jest w górze, zamiast ustawiania sygnału *swait* od razu, następuje zapis ′1′ do rejestru *wait_next*, a *swait* jest ustawiany w następnym takcie.

Konieczność obsługi osobnych przypadków dla zapisów i odczytów wynika z opóźnienia jakie wprowadza pamięć RAM. Przy zapisie potrzeba wystawić daną i adres równocześnie, a na początku następnego taktu nastąpi zapis. Przy odczycie natomiast adres należy podać takt przed odczytem wartości. W momencie, gdy zarówno blok *Comm* jak i *Mix* próbują zapisać dane do pamięci, wysyłany jest *swait* do *Mix* w tym samym takcie, żeby *Mix* podtrzymał stan wyjść bez zmian przez czas trwania następnego taktu zegara, ponawiając wtedy próbę zapisu. Natomiast w przypadku gdy *Comm* próbuje dokonać zapisu, a *Mix* następnego odczytu, dane potencjalnie zlecone do odczytu takt temu, są w bieżącym takcie poprawne i *Mix* powinien móc je odczytać. Ponieważ założono, że *Comm* ma priorytet, to zostanie dokonana operacja zapisu, pod wskazany przez *Comm* adres. W związku z tym, na wyjściu pamięci BRAM w przyszłym takcie pojawią się dane zapisane przez *Comm*, a nie te, które próbował odczytać *Mix*. W tym samym takcie

zostanie ponownie ustawiony adres odczytu, a takt później pojawią się odpowiednie dane na wyjściu pamięci. Dzięki opóźnieniu o 1 takt *swait*, *Mix* zignoruje dane, których nie powinien dostać, a takt później odczyta poprawnie dane.

```
case r.state is
  when ST_INIT =>
    r_n       <= REGS_INIT;
    r_n.state <= ST_LISTEN;
  when ST_LISTEN =>
    if swrite_comm = '1' then
      r_n.buf_in    <= r.buf_in(63 downto 0) & in_data;
      r_n.slice_num <= r.slice_num + 1;
      if r.slice_num = 3 then
        wait_flag := '1';
        if r.cnt_mix_trigger = 0 then
          c.swait <= '1';
        else
          r_n.wait_next <= '1';
        end if;
        c.addr_a <= std_logic_vector(r.cnt_words(5 downto 0));
        c.data_a <= r.buf_in(63 downto 32) &  -- "2"
                    r.buf_in(95 downto 64) &  -- "1"
                    in_data &                 -- "4"
                    r.buf_in(31 downto 0);    -- "3"
        c.wren_a <= '1';
        r_n.cnt_words <= r.cnt_words + 1;
        r_n.slice_num <= (others => '0');
      end if;  -- r.slice_num = 3
    ...
    end if; -- swrite_comm = '1'
    if r.wait_next = '1' then
      c.swait       <= '1';
      r_n.wait_next <= '0';
    end if;
  ...
  when others => null;
end case;
```

Listing 4.3. Fragment kodu VHDL bloku *reg* ilustrujący zszywanie i zapis danych wejściowych oraz generowanie sygnału *swait*.

Napływające dane wejściowe trafiają do rejestru przesuwnego *buf_in*. Za każdym razem jak napłynie 32-bitowe słowo, zwiększany o 1 jest licznik plastrów. Gdy jego wartość wyniesie 3, następuje sklejenie słów z rejestru przesuwnego ze słowem z wejścia i zapis do pamięci pod następny adres. Przy każdym zapisie, licznik słów jest zwiększany o 1.
Gdy wartość na 3 najmłodszych, spośród wszystkich 6, bitach licznika słów wynosi 7, a wartość licznika plastrów wynosi 3, to w pamięci jest wiersz gotowy do poddania operacji mieszania.

Operacja inicjalizowana jest poprzez przygotowanie licznika:
`r_n.cnt_mix_trigger <= to_unsigned(4, r.cnt_mix_trigger'length);`
Przez następne 4 takty, z prawdopodobną dodatkową przerwą w trakcie, będzie następował odczyt wierszy do mieszania. Numerowi wiersza odpowiada wartość na 3 najstarszych bitach licznika.

```
case to_integer(r.cnt_mix_trigger) is
  when 4 =>
    mix_addr := r.cnt_mix(5 downto 3) & "000";
    c.start  <= '1';
  when 3 =>
    mix_addr := r.cnt_mix(5 downto 3) & "010";
  when 2 =>
    mix_addr := r.cnt_mix(5 downto 3) & "110";
  when 1 =>
    mix_addr := r.cnt_mix(5 downto 3) & "100";
  when others =>
    mix_addr := (others => '0');
end case;
c.addr_a <= std_logic_vector(mix_addr);
c.addr_b <= std_logic_vector(mix_addr+1);
```

Listing 4.4. Sposób wyznaczania adresów odczytu danych do mieszania.

W kolejnych taktach odczytu danych, przy tworzeniu adresu, za tymi bitami doszywane są różne 3-bitowe stałe gwarantujące odczyt z przeplotem w zadanej kolejności.
Zapisy następują już bez dodatkowego przeplotu. Zatem powracające z mieszania wyniki są zapisywane pod kolejne adresy, wyznaczane na podstawie stanu licznika powracających słów *cnt_mix*.

W drugiej rundzie dane do *Mix* są wysyłane kolumnami. Zatem tym razem adresy budowane są poprzez zszycie stałych jako najstarszych bitów, a 3 najmłodszych bitów licznika jako najmłodszych bitów docelowej wartości.
Zapisy wyników drugiej rundy muszą się odbywać kolejno kolumnami. W związku z tym, adres wyznaczany jest jako rotacja wartości licznika *cnt_mix* o połowę swojej szerokości, czyli 3 bity.

Podczas odczytu w stanie *FLUSH*, odczytywane z pamięci są słowa o szerokości 128 bitów, ale do bloku *Comm* przekazywane są kolejne, 32-bitowe plastry. Starsze 6 bitów licznika słów ustawia adres pamięci, natomiast młodsze 2 są wykorzystywane do tego, aby wykorzystując multiplekser wybrać odpowiedni plaster ze słowa na wyjściu pamięci. Po każdym odczytanym plastrze, wartość licznika słów jest zwiększana o 1.

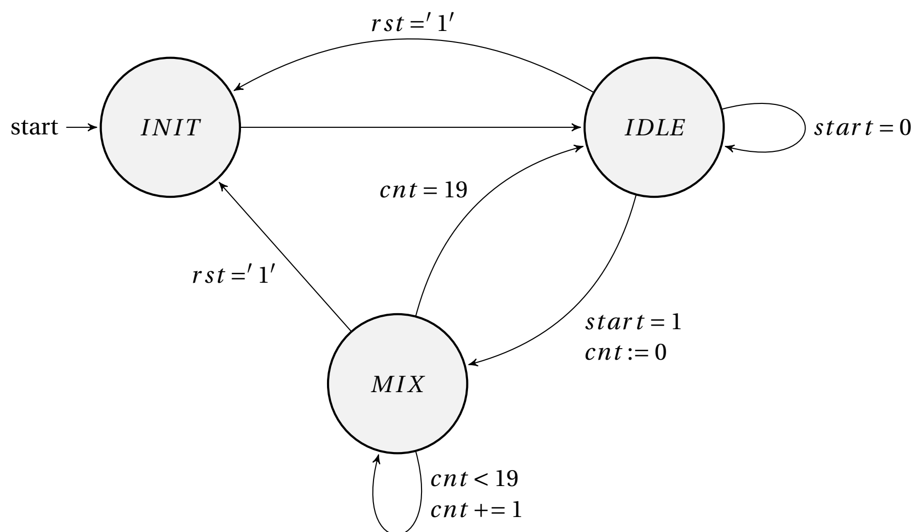
4.4. Blok Mix

Blok Mix realizuje nieznacznie zmodyfikowaną funkcję mieszania. Po otrzymaniu sygnału *start* przechodzi do stanu *MIX* i zeruje dodatkowy licznik kroków mieszania.

Przez 4 następne takty otrzymuje na wejściach kolejne fragmenty mieszanego wiersza lub kolumny, równolegle już wykonując pierwsze obliczenia. W 18. takcie mieszania blok zaczyna zwracać wyniki. Na wyjścia danych podaje pierwszą ćwiartkę obliczanego wektora oraz ustawia bit *swrite* na ′1′. Wyniki przekazuje do bloku *Reg*, za każdym razem dodatkowo zapalając bit *swrite*. Po zakończeniu obliczeń powraca do stanu *IDLE*.

W stanie *MIX* dla wartości licznika takich, że: $cnt < 4$ lub $cnt > 15$, podniesienie sygnału *swait* przez blok *Reg* powoduje zatrzymanie działania bloku *Mix*. Mechanizm ten zapewnia synchronizację dostępu do pamięci BRAM wewnątrz *Reg* pomiędzy blokami *Mix* oraz *Comm*.

Uproszczony, tzn. pomijający obsługę chwilowego zatrzymywania podczas transferu danych, graf automatu przedstawiono na rys.4.4.



Rysunek 4.4. Uproszczony graf stanów bloku *Mix*

Modyfikacje względem oryginalnie opisanej funkcji mieszania polegały na zrównolegleniu niezależnych operacji oraz zorganizowaniu ich w taki sposób, aby przesyłać możliwie najmniejszą część danych w ciągu pojedynczego taktu zegara, a co za tym idzie zmniejszyć szerokość interfejsów łączących bloki, bez jednoczesnego opóźniania wykonywania pierwszych operacji na tych danych.

Na proces mieszania składa się wielokrotne złożenie dwóch operacji opisanych pseudokodem:

- `fBlaMka(x,y)  := x+y+shift_left(x(31 downto 0)*y(31 downto 0), 1)`
- `xorrot(x,y,n) := (x xor y) ror n`

Obie operacje zaimplementowano jako funkcje o argumentach oraz zwracanej wartości typu `unsigned(63 downto 0)`. Napływające dane można kolejno przypisać do 64-bitowych zmiennych: A_1, A_2, A_3, A_4, B_1, ... C_1, ... D_1, ... D_4. W pierwszym takcie napływają wszystkie A, drugim - B, trzecim - D, a czwartym - C. Sekwencja w kolejnych taktach przebiega w

sposób następujący, gdzie $i = 1..4$, indeksy są liczone $mod4$, a in_i oraz out_i oznaczają kolejne porty wejściowe i wyjściowe:

1. $A_i = in_i$
2. $A_i = fBlaMka(A_i, B_i)$, gdzie $B_i = in_i$
3. $D_i = xorrot(D_i, A_i, 32)$, gdzie $D_i = in_i$
4. $C_i = fBlaMka(C_i, D_i)$, gdzie $C_i = in_i$
5. $B_i = xorrot(B_i, C_i, 24)$
6. $A_i = fBlaMka(A_i, B_i)$
7. $D_i = xorrot(D_i, A_i, 16)$
8. $C_i = fBlaMka(C_i, D_i)$
9. $B_i = xorrot(B_i, C_i, 63)$
10. $A_i = fBlaMka(A_i, B_{i+1})$
11. $D_i = xorrot(D_{i+3}, A_i, 32)$
12. $C_i = fBlaMka(C_{i+2}, D_i)$
13. $B_i = xorrot(B_i, C_i, 24)$
14. $A_i = fBlaMka(A_i, B_i)$
15. $D_i = xorrot(D_i, A_i, 16)$
16. $C_i = fBlaMka(C_i, D_i)$
17. $B_i = xorrot(B_i, C_i, 63)$,
    $out_i = A_i$
18. $out_i = B_{i+3}$
19. $out_i = C_{i+2}$
20. $out_i = D_{i+1}$

Całość zajmuje 20 taktów zegara.

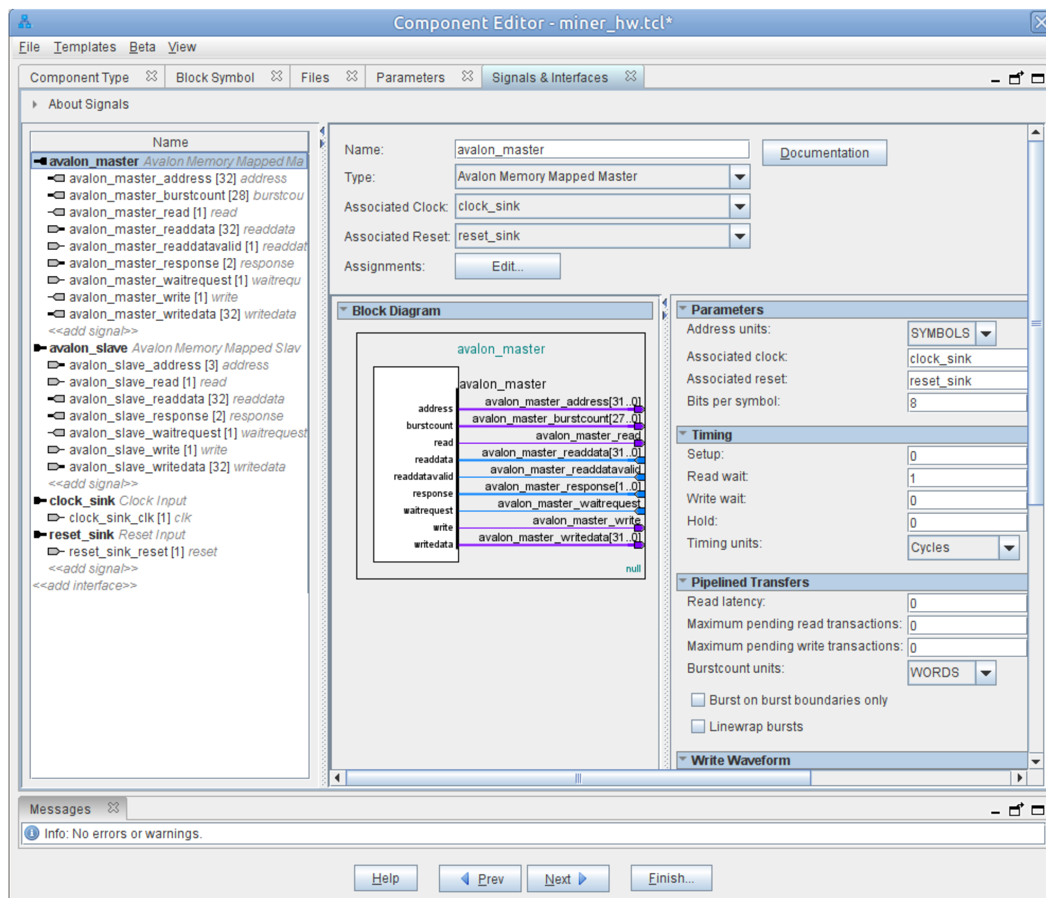
Biorąc pod uwagę fakt, że podczas pierwszej rundy mieszania równocześnie napływają do bloku *Reg* dane z *Comm*, należy dla najgorszego przypadku doliczyć dodatkowe 2 takty przerwy w trakcie pracy. Jeden podczas wczytywania danych do bloku *Mix*, w ciągu 4 pierwszych taktów. Drugi podczas zwracania wyników, w ciągu 4 ostatnich taktów. Oznacza to, że przez przynajmniej 10 kolejnych taktów zegara, blok będzie biernie oczekiwał na kolejny zestaw danych. Można by rozważyć redukcję zużycia zasobów sprzętowych potrzebnych do realizacji tego bloku. Przykładowo można by funkcję $fBlaMka$ liczyć w ciągu 2 taktów, a nie jednego. Spowodowało by to wydłużenie czasu obliczeń dla pojedynczego zestawu danych o 8 taktów, w zamian potencjalnie pozwoliło by to zmniejszyć zużycie bloków DSP dostępnych w układzie o 4. Natomiast powodowało by to wydłużenie drugiej rundy, czyli mieszania kolumn, o 64 takty zegara, a tymczasem to minimalizacja czasu przetwarzania danych jest głównym kryterium optymalizacji implementacji.

4.5. Połączenie koprocesora z HPS

Punkt wyjścia stanowił projekt referencyjny udostępniony w sieci przez producenta płyty, tzw. *GHRD* (Golden Hardware Reference Design) [33]. Po zaimportowaniu plików do narzędzia *Platform Designer*, usunięto wszystkie zbędne bloki z systemu. Pozostawiono tylko:

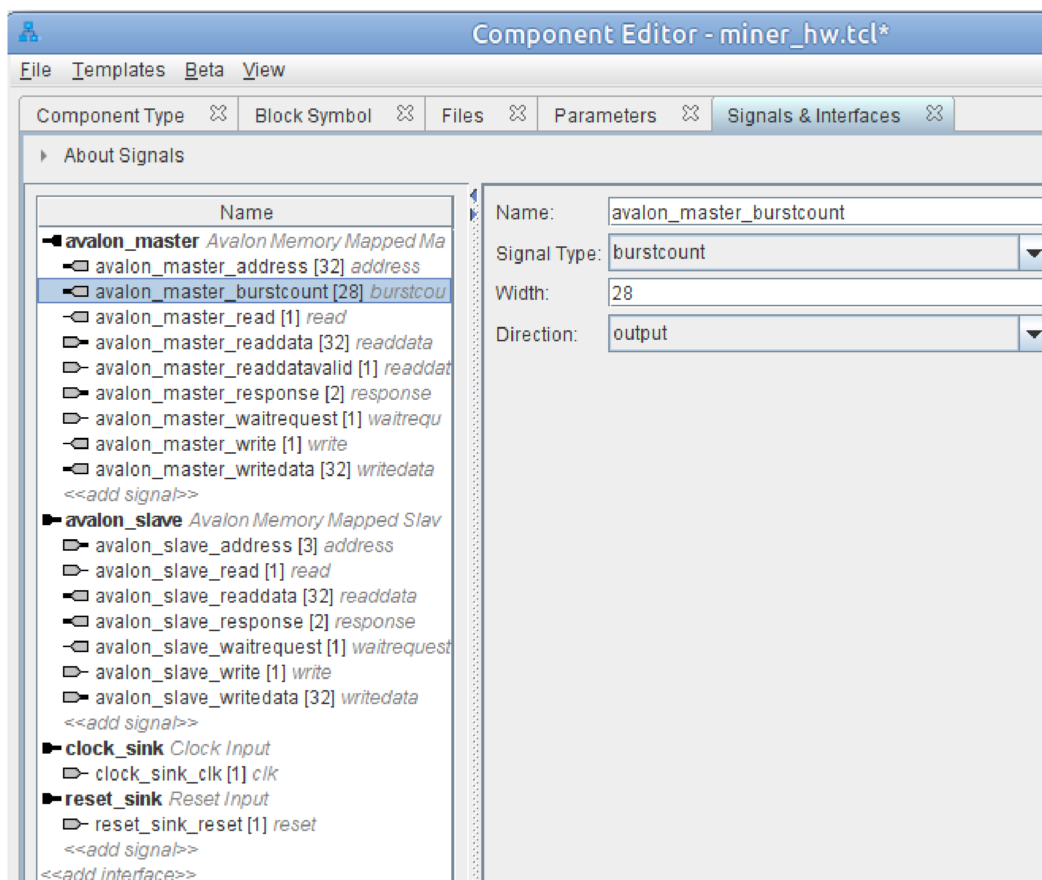
- HPS,
- źródło zegara,
- *System ID Peripheral*.

HPS (*Hard Processor System*) to komponent umożliwiający podłączenie układu programowalnego do procesora ARM wewnątrz czipu. *System ID Peripheral* to natomiast rdzeń IPCore, którego obecność w systemie jest wymagana przez producenta układu [34]. Przechowuje numer identyfikacyjny wersji systemu ustalany przez narzędzia podczas kompilacji systemu. Umożliwia narzędziom automatyczną kontrolę i rozpoznawanie wersji systemu.

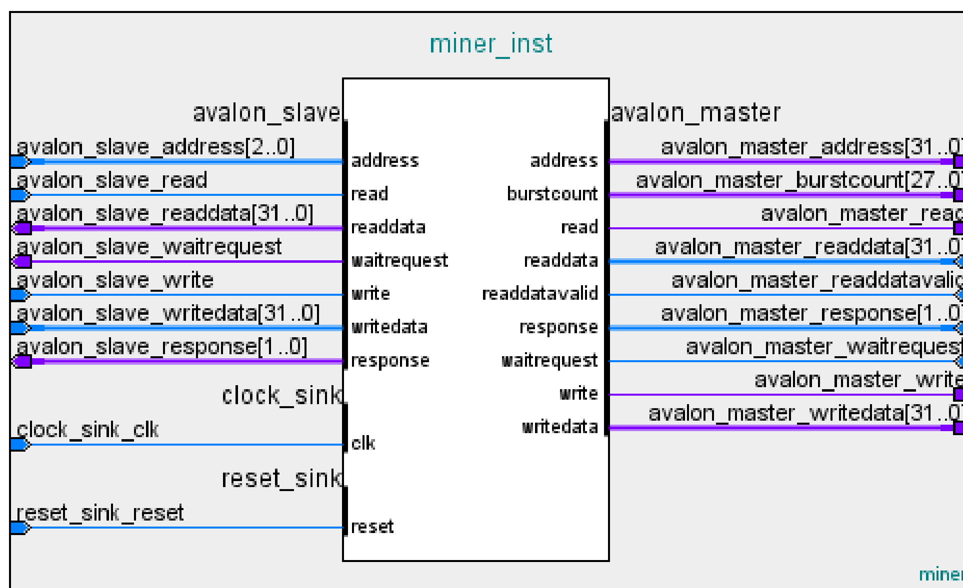


Rysunek 4.5. Okno kreatora komponentu, definicje portów

W pierwszej kolejności stworzono wrapper VHDL, to znaczy napisano pojedynczy plik, w którym stworzono pojedynczą instancję zawierającą wszystkie instancje bloków funkcjonalnych składających się na koprocesor wraz z połączeniami między nimi.



Rysunek 4.6. Fragment okna kreatora komponentu, definicje sygnałów



Rysunek 4.7. Kreator komponentu, podgląd komponentu

Następnie stworzono w kreatorze komponentów nowy typ komponentu co ilustrują rys.4.5, rys.4.6 oraz rys.4.7.

Zadeklarowanie typu komponentu polega na:

- podaniu nazwy oraz wyświetlanej nazwy komponentu,
- przypisaniu go do jednej z predefiniowanych grup,
- wskazaniu na pliki ze źródłami do syntezy i symulacji,
- podaniu parametrów, jeśli są wykorzystywane,
- określeniu i skonfigurowaniu portów oraz sygnałów składających się na nie.

Po utworzeniu komponentu, ręcznie zmodyfikowano wygenerowany plik *miner_hw.tcl*, rozszerzając go o przypisania skutkujące wygenerowaniem flag *compatible*, *group* oraz *vendor* w pliku drzewa urządzeń [35]. Na koniec utworzono połączenia między blokami oraz przydzielono adresy, co obrazuje rys.4.8.

Use	Connections	Name	Description	Export	Clock	Base	End	IRQ
✔		**hps_0**	Arria V/Cyclone V Hard Processor ...					
		f2h_cold_reset_req	Reset Input	**hps_0_f2h_cold_reset_...**				
		f2h_debug_reset_r...	Reset Input	**hps_0_f2h_debug_rese...**				
		f2h_warm_reset_req	Reset Input	**hps_0_f2h_warm_reset...**				
		f2h_stm_hw_events	Conduit	**hps_0_f2h_stm_hw_ev...**				
		memory	Conduit	**memory**				
		hps_io	Conduit	**hps_0_hps_io**				
		h2f_reset	Reset Output	*Double-click to export*				
		h2f_axi_clock	Clock Input	*Double-click to export*	**clk_0**			
		h2f_axi_master	AXI Master	*Double-click to export*	[h2f_axi_cl...			
		f2h_axi_clock	Clock Input	*Double-click to export*	**clk_0**			
		f2h_axi_slave	AXI Slave	*Double-click to export*	[f2h_axi_cl...	**0x0000_0000**	0xffff_ffff	
		h2f_lw_axi_clock	Clock Input	*Double-click to export*	**clk_0**			
		h2f_lw_axi_master	AXI Master	*Double-click to export*	[h2f_lw_axi...			
		f2h_irq0	Interrupt Receiver	*Double-click to export*		IRQ 0	IRQ 31	
		f2h_irq1	Interrupt Receiver	*Double-click to export*		IRQ 0	IRQ 31	
✔		**sysid_qsys**	System ID Peripheral Intel FPGA IP					
		clk	Clock Input	*Double-click to export*	**clk_0**			
		reset	Reset Input	*Double-click to export*	[clk]			
		control_slave	Avalon Memory Mapped Slave	*Double-click to export*	[clk]	0x0001_0000	0x0001_0007	
✔		**clk_0**	Clock Source					
		clk_in	Clock Input	**clk**	***exported***			
		clk_in_reset	Reset Input	**reset**				
		clk	Clock Output	*Double-click to export*	clk_0			
		clk_reset	Reset Output	*Double-click to export*				
✔		**miner_0**	miner					
		avalon_master	Avalon Memory Mapped Master	*Double-click to export*	[clock_sink]			
		avalon_slave	Avalon Memory Mapped Slave	*Double-click to export*	[clock_sink]	**0x0000_0000**	0x0000_001f	
		clock_sink	Clock Input	*Double-click to export*	**clk_0**			
		reset_sink	Reset Input	*Double-click to export*	[clock_sink]			

Rysunek 4.8. Kreator komponentu, gotowy system

Po syntezie i implementacji układu w narzędziu *Quartus* wygenerowno plik *.rbf* zawierający *bitstream* służący do zaprogramowania części programowalnej układu.

4.6. Sterownik urządzenia dla systemu Linux

Sterownik napisano w języku C, jako moduł jądra [36]. Zaimplementowano szereg standardowych funkcji umożliwiających obsługę koprocesora jako urządzenia znakowego. Stworzono strukturę przedstawioną na listingu 4.5 opisującą rejestry urządzenia zgodnie z tab.4.2.

```c
typedef struct device_registers {
  uint32_t id;
  uint32_t ctrl;
  uint32_t status;
  uint32_t addr;
  uint32_t len;
} dev_regs;
```

Listing 4.5. Struktura opisująca rejestry obsługiwanego urządzenia

Wymiana danych z koprocesorem następuje poprzez współdzielony bufor DMA. Podczas rejestracji sterownika wywoływana jest funkcja `probe_miner()`. Po utworzeniu urządzenia, funkcja ta zleca systemowi przydzielenie ciągłego bufora w pamięci RAM poprzez wywołanie funkcji `dmam_alloc_coherent()` z odpowiednim zestawem argumentów. Na sam koniec funkcja `probe_miner()` dokonuje odczytu zawartości rejestru `ID` koprocesora i porównuje odczytaną wartość ze stałą zdefiniowaną w kodzie sterownika. Umożliwia to kontrolę wersji oraz wykrycie ewentualnych rozbieżności wynikających z błędów przy wgrywaniu obrazów systemu i konfiguracji układu programowalnego. Mechanizm ten można również w przyszłości wykorzystać do implementacji różnego sposobu obsługi różnych wersji koprocesora. W przypadku wystąpienia jakiejkolwiek różnicy, logowany jest komunikat o błędzie przy pomocy funkcji `dev_err`, będącej nakładką na funkcję `printk` z poziomem dziennika `KERN_ERR` dedykowaną dla sterowników urządzeń [37]. Następnie procesor przechodzi do procedury obsługi błędu `err1`, w której zwalniana jest pamięć jeśli została zaalokowana, a urządzenie jest odrejestrowywane.

```
if(dregs->id != DEVICE_ID) {
    dev_err(&pdev->dev, "Incorrect ID of %s\n"
        "ID: %4.4lx\n"
        "Expected: %4.4lx\n", DEVICE_NAME, dregs->id, DEVICE_ID);
    goto err1;
```

Listing 4.6. Sprawdzenie odczytanego numeru *ID* urządzenia.

Funkcja `mmap_miner()` umożliwia przetłumaczenie fizycznego adresu bufora na wirtualny, z którego potem korzystają aplikacje.

```
int mmap_miner(struct file *filp,
              struct vm_area_struct *vma)
{
  int remap=0;
  unsigned long physical = virt_to_phys(fdata);
  unsigned long vsize = vma->vm_end - vma->vm_start;
  unsigned long psize = ALLOC_SIZE;
  dev_info(&my_pdev->dev, "mmap, physaddr: %lx\n",physical);
  if(vsize>psize)
    return -EINVAL;
  vma->vm_page_prot = pgprot_noncached(vma->vm_page_prot);
  remap=dma_mmap_coherent(&my_pdev->dev,vma,fdata, dma_addr, vsize);
  dev_info(&my_pdev->dev, "remap = %d\n", remap);
  if (vma->vm_ops)
    return -EINVAL;
  vma->vm_ops = &vm_ops_miner;
  vma_open_miner(vma);
  return 0;
}
```

Listing 4.7. Funkcja `mmap_miner`.

Procedury odczytu oraz zapisu rejestrów koprocesora implementuje `ioctl_miner()`. Funkcja ta jest pojedynczą instrukcją wielokrotnego wyboru `switch`, gdzie kolejnym przypadkom, odpowiadają kolejne numery komend. W przypadku wywołania funkcji z błędnym numerem komendy zwracana jest stała `-EINVAL` informująca o błędzie. Kod komendy tworzony jest automatycznie jako wartość makra `_IO()`, które przyjmuje 2 argumenty: tzw. *magiczną liczbę*, będącą 8-bitową liczbową, unikatową dla każdego sterownika oraz numer komendy. Przykładową definicję takiego kodu komendy przedstawiono poniżej:

```
#define MINER_IOC_READ_STATUS _IO(MINER_IOC_MAGIC, 0xad)
```

Na odczyt rejestru składają się 2 operacje:

1. bariera pamięci [38] [39],
2. zwrócenie wartości rejestru, czyli odpowiedniego pola struktury typu `device_register`.

Na zapis do rejestru składają się:

1. zapis wartości do rejestru, czyli przypisanie argumentu do odpowiedniego pola struktury typu `device_register`,
2. bariera pamięci,
3. zwrócenie stałej SUCCESS.

Zastosowanie barier typu `general` wymusza oczekiwanie na zakończenie wszystkich operacji odczytu i zapisu do pamięci, co gwarantuje jej spójność. Pozwala uniknąć problemów związanych z wykonywaniem poza kolejnością.

```
long ioctl_miner(struct file *filp, unsigned int cmd, unsigned long arg)
{
  switch(cmd) {
    case MINER_IOC_READ_STATUS: {
      mb();
      return dregs->status;
    }
    ...
    case MINER_IOC_WRITE_CTRL: {
      dregs->ctrl = arg;
      mb();
      return SUCCESS;
    }
    ...
    default:
      return -EINVAL;
  }
}
```

Listing 4.8. Funkcja `ioctl_miner()`, przykładowa obsługa operacji odczytu lub zapisu wartości do rejestru koprocesora.

4.7. Modyfikacje kodu referencyjnego

Przygotowano w sumie 4 wersje programu:

- implementację referencyjną,
- implementację akcelerowaną koprocesorem,
- implementację referencyjną, gdzie funkcję `XORBlock` zastąpiono instrukcją `veor_u64` wykorzystującą rozszerzenie *Neon*, a do flag kompilatora dodano `-mfpu=neon`,
- implementację wykorzystującą zarówno koprocesor jak i rozszerzenie *Neon*.

W celu dodania obsługi koprocesora wprowadzono w kod szereg modyfikacji. Dodano niezbędne do wykorzystania sterownika pliki nagłówkowe *fcntl.h*, *sys/mman.h* i *miner_ioc_cmds.h* oraz zewnętrzną zmienną `extern int fminer` zawierającą deskryptor urządzenia. W pliku *Test / argon2-test.c* dodano procedurę inicjalizacji koprocesora. Program w pierwszej kolejności otwiera urządzenie w taki sam sposób jak plik. Następnie uzyskuje wskaźnik na bufor DMA. Restartuje koprocesor, zapisuje w rejestrach sterujących adres fizyczny początku bufora i długość bloku danych wyrażoną w słowach 32-bitowych korzystając z komend `ioctl` sterownika. Funkcja obsługująca komendy ioctl w sterowniku ignoruje trzeci argument funkcji `ioctl`, kiedy drugim jest numer komendy `MINER_IOC_WRITE_ADDR`. Sterownik w tym przypadku zapisuje w rejestrze urządzenia odpowiadającym za adres początku bufora wartość przechowywaną w jednej ze swoich zmiennych, niedostępnej dla aplikacji użytkownika:
`dregs->addr = dma_addr;`
W związku z tym, trzeci argument wywołania `ioctl` może przyjmować dowolną wartość typu `unsigned long`. W celu podniesienia czytelności kodu oraz podkreślenia faktu, że ta wartość nie ma znaczenia, zastosowano makro `NULL`.

```
fminer=-1;
fminer=open("/dev/miner0", O_RDWR);
if(fminer==-1)
{
  perror("/dev/miner0");
  printf("I can't open device!\n");
  fflush(stdout);
  exit(1);
}
//Map data buffer
volatile const uint32_t * buf = (uint32_t *) mmap(0,0x1000,PROT_READ |
    PROT_WRITE,MAP_SHARED,fminer,0x0);
if(buf == (void *) -1l)
{
  perror("Can't map data buffer!\n");
}

ioctl(fminer, MINER_IOC_WRITE_CTRL, 2); // Reset
usleep(1);
```

```
ioctl(fminer, MINER_IOC_WRITE_ADDR, NULL); // Write dma buffer addr
ioctl(fminer, MINER_IOC_WRITE_LEN, 256); // Write block len
```

Listing 4.9. Procedura inicjalizacji koprocesora w pliku *Test / argon2-test.c*.

Do zestawów argumentów różnych funkcji w różnych plikach, wywoływanych po kolei w głąb, od `Run` do `FillBlock`, dodano na końcu ulotny wskaźnik na bufor. W pliku *Argon2 / argon2-ref-core.c* 2 rundy algorytmu *Blake2* realizowane w HPS zastąpiono: przygotowaniem danych w buforze, uruchomieniem, aktywnym oczekiwaniem na zakończenie pracy koprocesora, wykorzystaniem danych z bufora do dalszych obliczeń i na końcu zatrzymaniem koprocesora. Zmiany w procedurze przygotowania danych przedstawiono na listingu 4.10. Wywołania `XORBlock` i `CopyBlock` zastąpiono pętlą `for`, w której następują xorowanie oraz kolejne przypisania, tak aby zminimalizować liczbę zbędnych kopiowań.

```
 void FillBlock(const block* prev_block, const block* ref_block,
-        block* next_block, const uint64_t* Sbox) {
+        block* next_block, const uint64_t* Sbox, volatile uint64_t* dma_buf) {
-    block blockR;
-    CopyBlock(&blockR,ref_block);
-    XORBlock(&blockR,prev_block);
     block block_tmp;
-    CopyBlock(&block_tmp, &blockR);
-
-    uint64_t x = 0;
-    if (Sbox != NULL) {
-        x = blockR.v[0] ^ blockR.v[ARGON2_WORDS_IN_BLOCK - 1];
-        for (int i = 0; i < 6 * 16; ++i) {
+    for(unsigned int i = ARGON2_WORDS_IN_BLOCK; i--; ){
+         dma_buf[i] = block_tmp.v[i] = ref_block->v[i] ^ prev_block->v[i];
+    }
+
+    ioctl(fminer, MINER_IOC_WRITE_CTRL, 1); // Start coproc
+    uint64_t x = (Sbox != NULL) ? (
+        block_tmp.v[0] ^ block_tmp.v[ARGON2_WORDS_IN_BLOCK - 1]) : 0;
+    if (x != 0) {
+        for (int i = 6 * 16; i--; ) {
```

Listing 4.10. Fragment łaty zawierającej zmiany w pliku *Argon2 / argon2-ref-core.c*.

Po zakończeniu obliczeń przez koprocesor i odczycie danych przez aplikację użytkownika na procesorze, następuje zapis wartości ′0′ do rejestru kontrolnego koprocesora. Powoduje to przejście koprocesora ze stanu *DONE* do *IDLE*, przy okazji powodując reset bloków obliczeniowych.

```
PAKIET_VERSION = 1.0
PAKIET_SITE    = $(BR2_EXTERNAL_NAZWA_TREE_PATH)/src/pakiet
PAKIET_SITE_METHOD = local
#PAKIET_LICENSE = LGPLv2.1/GPLv2
PAKIET_DEPENDENCIES += fpga-module linux

define PAKIET_BUILD_CMDS
   $(MAKE) $(TARGET_CONFIGURE_OPTS) all -C $(@D)
endef
define PAKIET_INSTALL_TARGET_CMDS
   $(INSTALL) -D -m 0755 $(@D)/pakiet $(TARGET_DIR)/usr/bin
endef

$(eval $(generic-package))
```

Listing 4.11. Przykładowa zawartość pliku .mk

4.8. Przygotowanie zewnętrznych pakietów

Przygotowano pakiety do wkompilowania w obraz systemu zgodnie zaleceniami zawartymi w instrukcji do narzędzia *Buildroot* [40]. Obok folderu z narzędziem utworzono nowy folder i zadeklarowano tam tzw. *br2-external tree*. Ustawiono wartość zmiennej programu *make* `BR2_EXTERNAL` na położenie tego folderu. Utworzono w nim 3 pliki:

- *external.desc*, zawierający deklarację nazwy drzewa `BR2_EXTERNAL`:
  ```
  name: NAZWA_TREE
  ```
- *external.mk*, o treści:
  ```
  include $(sort $(wildcard \
      $(BR2_EXTERNAL_NAZWA_TREE_PATH)/package/*/*.mk))
  ```
- *Config.in*, który zawiera ścieżki na pliki Config.in poszczególnych pakietów.

Utworzono dodatkowo 3 kolejne foldery *package*, *src* oraz *rootfs_overlay*.

W folderze *package* na każdy pakiet dodany do *br2-external tree* przypada folder zawierający 2 pliki: *Config.in* oraz plik *.mk*. *Config.in* zawiera, wcięte o 1 tabulator, kolejno:

- słowo kluczowe `config`, a za nim zmienną `BR2_PACKAGE_PAKIET`,
- flagę `bool "pakiet"`,
- listę zależności, np:
  ```
  depends on BR2_LINUX_KERNEL
  select BR_FPGA_MODULE
  ```
- tekst pomocy poprzedzony słowem kluczowym `help`, wcięty dodatkowo o 2 spacje,
- komentarz poprzedzony słowem kluczowym `comment`.

Przykładowy plik *pakiet.mk* zamieszczono poniżej na listingu 4.11.

W folderze *src* natomiast umieszczono kody źródłowe programu z danego pakietu oraz standardowy *Makefile*.

Poza pakietami przygotowano również tzw. *rootfs_overlay* [40], czyli nakładkę z drzewem plików kopiowanym na samym końcu procesu budowania do systemu plików.

```
set -e
PATH=/root:/bin:/usr/bin:/sbin:/usr/sbin:/usr/local/sbin
case "$1" in
start)
mount /dev/mmcblk0p3 /mnt
cd /mnt
udhcpc
modprobe miner

;;
stop)
rmmod miner
sync
;;
esac
exit 0
```

Listing 4.12. Zastosowana w projekcie nakładka na *rootfs* z dodatkową procedurą init.

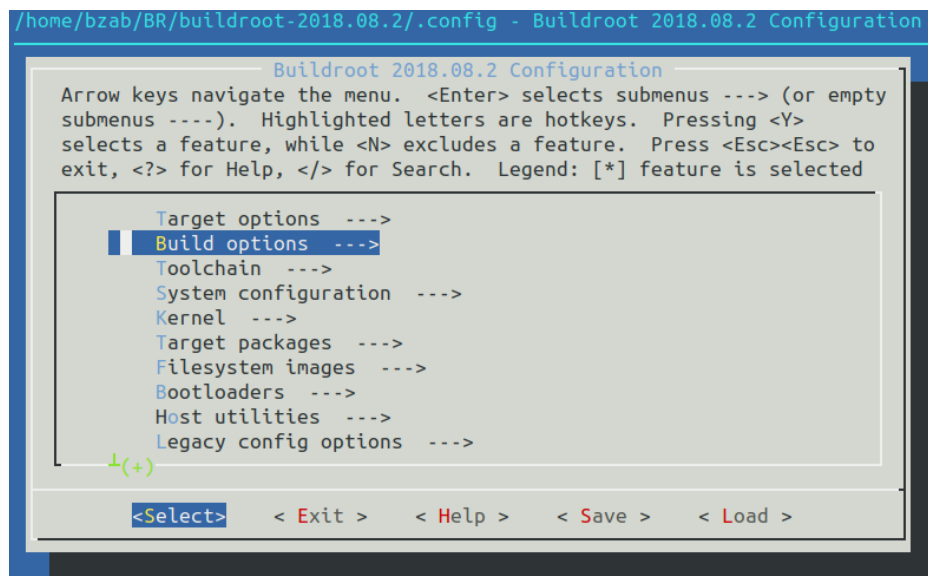
W tym przypadku była to nakładka z dodatkową procedurą *init*. W folderze *rootfs_overlay* utworzono plik *etc / S90custominit* o treści przedstawionej na listingu 4.12.
Procedura ta po uruchomieniu systemu automatycznie montowała kartę SD, uruchamiała klienta DHCP oraz sterownik koprocesora. Natomiast przed zamknięciem systemu usuwała moduł koprocesora, a komenda `sync` synchronizowała pamięć. *S90* na początku nazwy powoduje, że skrypt ten będzie jednym z ostatnich wykonywanych po uruchomieniu systemu, ponieważ uruchamiane są w kolejności od *S00* do *S99* [41].

4.9. Przygotowanie obrazu systemu operacyjnego

Pobrano dostępny publicznie na grupie dyskusyjnej *Google alt.sources* [42] zestaw plików dodający wsparcie dla płyty *DE0-Nano-SoC* i dodano te pliki do narzędzia *Buildroot*.

Następnie przygotowano drzewo urządzeń. W tym celu wygenerowano w *Platform Designer* plik *soc_system.sopcinfo*. Skopiowano z przykładowych projektów udostępnionych przez producenta płyty pliki *soc_system_board_info.xml* oraz *hps_common_board_info.xml*. W pliku *soc_system_board_info.xml* wprowadzono niezbędne modyfikacje, umożliwiające wykorzystanie obydwu rdzeni procesora ARM oraz wyłączające nieużywane peryferia. Następnie wykorzystano program *sopc2dts* dostępny w ramach środowiska *intelSoCEDS*, by na podstawie pliku *.sopcinfo* oraz obydwu plików *board_info* wygenerować plik drzewa urządzeń *.dts*.

Przy pomocy polecenia `make menuconfig` uruchomionego w katalogu z narzędziem *Buildroot* uruchomiono edytor konfiguracji.
W zakładce *Target options* wybrano architekturę *ARM (little endian)*, wariant *cortex-A9* oraz zestaw instrukcji *ARM*. Format plików binarnych ustawiono na *ELF*. Włączono obsługę rozszerzeń *NEON SIMD* oraz *VFP*.
W trakcie profilowania oraz testów wykorzystywano różne poziomy *gcc debug level* oraz *gcc optimization level* dostępne po włączeniu opcji *build packages with debugging symbols*



Rysunek 4.9. Przykładowy zrzut ekranu z narzędzia *Buildroot*.

oraz *strip target binaries* w zakładce *Build options*.
W zakładce *System configuration* wybrano *BusyBox* jako *Init system*. W celu umożliwienia bezproblemowego, zdalnego dostępu do systemu przez SSH dodano hasło dla użytkownika *root*. Wskazano ścieżkę na folder z nakładkami na *rootfs*. Podano ścieżkę na skrypt *genimage.sh* do uruchomienia po utworzeniu systemu plików oraz argumenty do przekazania przy uruchamianiu tego skryptu: `-c board/altera/de0_nano/genimage.cfg`.
W zakładce *Kernel* podano nazwę pliku defconfig, wybrano *zImage* jako format obrazu, ustawiono rodzaj kompresji *gzip*. Włączono budowanie pliku *.dtb* przez narzędzie *Buildroot* i wskazano ścieżkę na plik drzewa urządzeń *.dts*.
W *Target packages* dodano pakiety z programami:

- *Dropbear*, umożliwiający dostęp do systemu poprzez SSH,
- *Ethtool*, wykorzystywane przy usuwaniu problemów związanych z interfejsem sieciowym,
- *Dosfstools*, *E2tools*, do obsługi różnych systemów plików takich jak: DOS, EXT2, EXT3,

W *Bootloaders* wybrano *U-Boot* jako *bootloader*. Zaznaczono, że wymaga *dtc*. Podano ścieżki na plik z obrazem środowiska oraz *defconfig* płyty.
W *External options* zaznaczono pakiet sterownika oraz pakiety z programami dla których mierzono czasy wykonania.

Przygotowano obraz karty SD, z której system będzie uruchamiany. Wgrano na kartę minimalny, działający obraz systemu, a następnie go skopiowano. Obraz systemu w miarę rozwoju będzie nadpisywany, natomiast kopia pozostanie bez zmian. Dzięki temu w przypadku wgrania niedziałającej wersji systemu podczas rozwoju, możliwe będzie przerwanie procedury startu systemu i następnie z poziomu konsoli programu *U-Boot*

ręczne załadowanie początkowych obrazów. Następnie po załadowaniu działających obrazów, możliwe będzie pobranie przez sieć nieco wcześniejszej, działającej wersji systemu docelowego, albo wgranie nowej, poprawionej wersji.

Po uruchomieniu minimalnego, działającego obrazu systemu na płycie, kolejne aktualizacje można wysyłać przez sieć. Po ukończeniu kompilacji i wygenerowaniu potrzebnych plików, tzn. aktualnych wersji:

- pliku *zImage* zawierającego obraz systemu wraz z *bootloaderem*,
- pliku *.rbf* zawierającego *bitstream* służący do zaprogramowania układu FPGA,
- drzewa urządzeń skompilowanego do binarnego pliku *.dtb*,

na maszynie, na której będzie stał serwer plików, należy przenieść pliki do 1 folderu. Następnie wywołać komendę: `python3 -m http.server`. Domyślnie serwer pracuje na porcie 8000. Po zalogowaniu się na systemie działającym na płycie wystarczy przy pomocy polecenia `wget` pobrać aktualne pliki i nadpisać poprzednie wersje. Po zsynchronizowaniu pamięci i zrestartowaniu płyty nowe obrazy powinny zostać załadowane.

5. Weryfikacja rozwiązania

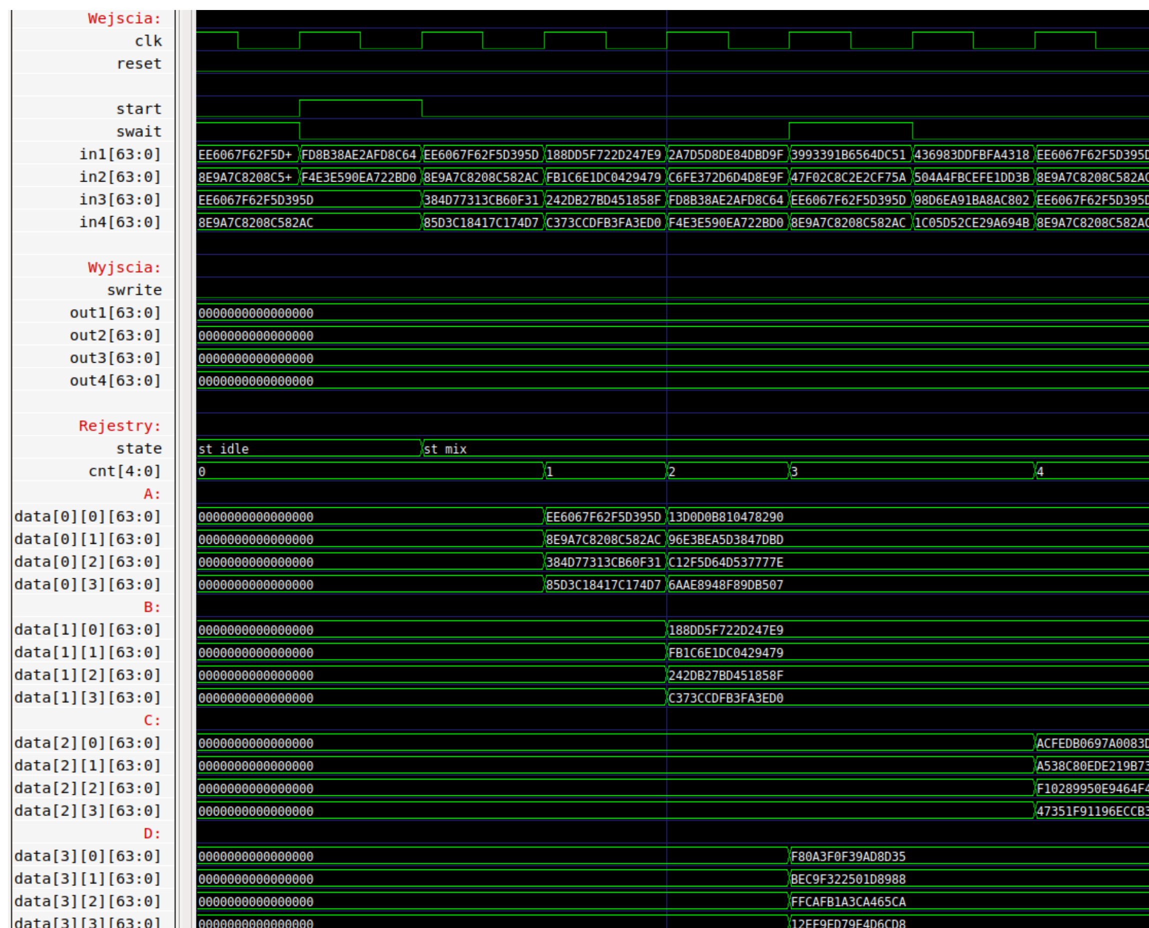
5.1. Weryfikacja koprocesora w symulacji

Wszystkie metody weryfikacji koprocesora zostały oparte na wybranym wcześniej fragmencie kodu referencyjnego przedstawionego na listingu 3.1 i nazywanego dalej **modelem w C**, który ma być realizowany przez koprocesor. Postanowiono w maksymalnym stopniu wykorzystać rozwiązania programowe o otwartych źródłach.

Na początku napisano zestaw skryptów, które na podstawie modelu w C lub jego fragmentów przy stosowanych na początku *testbenchach* dla pojedynczych bloków funkcjonalnych, generowały pary plików tekstowych: jeden z wektorami wejściowymi, drugi z referencyjnymi wektorami wyjściowymi. Wektory wejściowe mogły być generowane w sposób pseudolosowy albo w celu ułatwienia ręcznej analizy mogły być predefiniowane. Stworzono *testbenche* w języku VHDL. Zawierały one proces, który wczytywał plik tekstowy z danymi wejściowymi do tablicy, a następnie, w kolejnych taktach zegara, na ich podstawie generował zestaw pobudzeń dla badanego bloku. Po uruchomieniu *testbencha* w symulatorze *GHDL*, w wersji 0.37, symulator generował plik z przebiegami w formacie *.gtk*, który można było następnie obejrzeć w programie *GTKWave*. Umożliwiało to ręczną analizę działania koprocesora lub bloków funkcjonalnych. Metoda ta była wystarczająca aby ocenić czy maszyna stanów wewnątrz układu działa poprawnie. Natomiast biorąc pod uwagę charakter zbliżony do szumu wyników pośrednich, jak i końcowych, ręczna analiza i ocena poprawności wyników była wysoce nieefektywna oraz stwarzała znaczne ryzyko wystąpienia pomyłek.

Następnie zmodyfikowano *testbenche* poprzez dodanie drugiego procesu, który zbierał dane wynikowe w tablicy, a na koniec testu eksportował je do pliku tekstowego. Tak otrzymane wyniki można było następnie porównywać z wektorami referencyjnymi przy pomocy zewnętrznych skryptów napisanych w dowolnym języku, np. Python czy dowolnym języku powłoki poleceń. Przy uruchamianiu pojedynczych testów nie zaobserwowano problemów z szybkością ich wykonywania. Znaczną wadą takiego podejścia do testowania jest konieczność opisania wszystkiego na poziomie pojedynczych sygnałów, bezpośrednio w kodzie *testbencha* VHDL. Skutkuje to znacznym poziomem skomplikowania, czasochłonnością, niską elastycznością oraz słabą przenaszalnością kodu pomiędzy kolejnymi projektami.

Ostatecznie zastąpiono *testbenche* pisane w VHDL **frameworkiem CocoTB** w wersji 1.4.0. Framework ten upraszcza weryfikację układów cyfrowych. Główny nacisk położony jest na weryfikację w warstwie transakcji, randomizowane testy i łatwe ponowne wykorzystanie plików projektowych. Testy przygotowano w języku Python, w wersji 3.8.0, w oparciu o dostępne w bibliotece klasy. Następnie przygotowano zestaw plików programu *Make*, wskazujących na moduł języka Python zawierającego fabrykę testów, budujących



Rysunek 5.1. Przykładowy zrzut ekranu z GTKWave

dynamicznie linkowaną bibliotekę z modelem w C oraz umożliwiających skompilowanie kodu VHDL i uruchomienie go w symulatorze.

Podczas implementowania dynamicznie tworzonych testów w *CocoTB* napotkano na szereg problemów:

1. ograniczoną liczbę przykładów ilustrujących wykorzystanie różnych narzędzi oferowanych przez framework,
2. niekompletną implementację standardu Avalon-MM w klasach interfejsów,
3. problem z instancjonowaniem rdzeni *IPCore* udostępnianych w ramach narzędzi producenta układu, bez dostępu do ich kodu źródłowego.

Skonfigurowano skrypty dostępne w ramach symulatora *GHDL* w sposób umożliwiający prekompilację elementów biblioteki producenta układu [43]. Następnie stworzono dodatkową regułę dla programu *make* w pliku *Makefile* do *CocoTB*, umożliwiającą skompilowanie bibliotek przed kompilacją całego projektu. Dodano ją do zależności głównej reguły budującej cały projekt. Do symulatora przekazano dodatkowe argumenty umożliwiające symulowanie tych rdzeni *IPCore*:

- `-P=altera`

wskazuje na folder z prekompilowanymi bibliotekami,

- `--ieee=synopsys`
  dodaje dodatkowe pakiety bibliotek spoza standardu IEEE [44],
- `-fexplicit`
  w przypadku konfliktu powodującego niejednoznaczność przeciążenia operatora umożliwia podanie w kodzie, z której biblioteki ma być brana definicja danego operatora w danym miejscu.

Dodatkowo, ponieważ rdzenie *IPCore* producenta nie zawsze mają inicjalizowane wartości wyjść oraz framework *CocoTB* domyślnie ustawia wartość niewykorzystywanych w danej chwili wyjść na $'X'$, dodano flagę `--ieee-asserts=disable`, powodującą ukrycie ostrzeżeń o wykryciu metawartości w symulacji.

Pierwsze próby uruchomienia *testbencha* w *CocoTB* kończyły się niepowodzeniem. Najpierw program zgłaszał błędy związane z nieistniejącym polem `byteenable` w instancjach klas driverów magistrali podłączonych do interfejsów symulowanego koprocesora. Specyfikacja interfejsów *Avalon-MM* [31] określa ten sygnał jako opcjonalny. W wykorzystywanej wersji *CocoTB*, zapisy i odczyty niepełnych słów nie są wspierane. Oznacza to, że w trakcie kosymulacji kod w języku Python przy transakcjach sprawdza czy wszystkie bity sygnału `byteenable` są równe $'1'$ i w przeciwnym przypadku zwraca błąd. W związku z tym zaimplementowano poprawkę w kodzie frameworku, powodującą że jeśli zostanie wykryty brak pola `byteenable`, to dalsze operacje są wykonywane w taki sposób, jakby to pole istniało i posiadało wartość odpowiadającą wektorowi jedynek.
Po naprawieniu tego błędu testy dalej nie działały, były przerywane w połowie, a interpreter Pythona podnosił wyjątek:

```
'Write to object {} was scheduled during a read-only sync phase.'
```

Przyczyną problemu był drobny błąd w implementacji obiektu imitującego pamięć podpiętą interfejsem *Avalon-MM*. Mianowicie w procedurze generowania przerw w transmisji, sygnalizowanych poprzez podniesienie sygnału `waitrequest`, nie we wszystkich przypadkach następowało oczekiwanie na następny delta cykl symulacji przed próbą przypisania nowej wartości do sygnału. Mogło to powodować próby zapisu w czasie fazy, kiedy dozwolony był tylko odczyt, a co za tym idzie błąd i przerwanie symulacji. Oba problemy zgłoszono w repozytorium projektu oraz zaproponowano łatę z wyżej opisanymi poprawkami [45].

Kolejnym problemem okazał się być brak generowania przerw w transmisji przy symulowanym odczycie z pamięci. Nie udało się w prosty sposób dodać tej funkcji do frameworku, w związku z czym zadecydowano o pominięciu testowania tego szczególnego przypadku w symulacji i pozostawienie weryfikacji testom przeprowadzanym na fizycznie zrealizowanym systemie.

Po uruchomieniu procedury testującej w oknie konsoli na bieżąco wyświetlano konfigurację każdego uruchamianego testu, a następnie jego wynik, co obrazuje rys.5.2. *Te-*

stbench zaimplementowano w taki sposób, że gdy nie otrzymał jakiegoś parametru, tzn. parametr miał przypisaną wartość `None`, to wartość parametru była losowana z dopuszczalnego zakresu. Na koniec wyświetlano w konsoli zbiorczy raport informujący o czasie symulowanym i rzeczywistym czasie trwania symulacji oraz liczbie nieudanych testów jak na rys.5.3.

```
7864360.00ns INFO     Starting test: "run_test_322"
                      Description: Automatically generated test

                        block: None
                        block_addr: 0
                        wait_req: False
                        wait_req_max_len: 3

7895040.00ns INFO     Results OK
7895140.00ns INFO     Test Passed: run_test_322
```

Rysunek 5.2. Przykładowy wynik pojedynczego testu w CocoTB

```
                     ** argon_fpga_tb.run_test_557    PASS       16940.00          0.53      32105.74  **
                     ** argon_fpga_tb.run_test_558    PASS       19740.00          0.51      38869.46  **
                     ** argon_fpga_tb.run_test_559    PASS       30940.00          0.66      46802.48  **
                     ** argon_fpga_tb.run_test_560    PASS       38060.00          0.83      45649.48  **
                     ***********************************************************************************

15748940.00ns INFO   ************************************************************************************
                     **                                ERRORS : 0                                      **
                     ************************************************************************************
                     **                          SIM TIME : 15748940.00 NS                             **
                     **                         REAL TIME : 318.07 S                                   **
                     **                   SIM / REAL TIME : 49514.33 NS/S                              **
                     ************************************************************************************

15748940.00ns INFO   Shutting down...
```

Rysunek 5.3. Przykładowe podsumowanie zestawu testów w CocoTB

W przypadku otrzymania błędnych wyników w symulacji, zwracany był błąd oraz wydrukowane w 2 kolumnach dane referencyjne i z symulacji, co ilustrują rys.5.4 oraz rys.5.5. Z niewiadomych przyczyn, w sytuacji gdy na procedurę składa się wiele przypadków testowych, w testach następujących po nieudanym teście dochodzi do zawieszenia się programu. Instancja klasy imitującej pamięć nie wystawia całego wektora danych w transakcji, tylko przerywa transakcję na 8 słów przed końcem, a następnie pozostaje bezczynna. Symulacja zostaje wtedy przerwana na skutek osiągnięcia limitu czasu.

```
17540.00ns ERROR    Test Failed: run_test_001 (result was TestFailure)
                    Traceback (most recent call last):
                      File "/home/bzab/.local/lib/python3.8/site-packages/cocotb/regression.py", line 596, in _my_test
                        await function(dut, *args, **kwargs)
                      File "/home/bzab/FPGA/fpga-argon2d-miner/fpga/tests/argon_fpga_tb.py", line 205, in run_test
                        await tb.data_test()
                      File "/home/bzab/FPGA/fpga-argon2d-miner/fpga/tests/argon_fpga_tb.py", line 149, in data_test
                        raise TestFailure(str_comp+"Received result differs from expected")
                    cocotb.result.TestFailure: R: 0x586f9a3e E: 0x5eb5c40f
                    R: 0xb09fa792 E: 0x7fc4b54e
                    R: 0x4b77354c E: 0xba0acdea
                    R: 0x7a7dc457 E: 0x3c321720
```

Rysunek 5.4. Przykładowy test zakończony błędnym wynikiem w CocoTB

```
                      R: 0xdb1206a1 E: 0x84d0123f
                      Received result differs from expected
17540.00ns ERROR      Failed 1 out of 1 tests (0 skipped)
17540.00ns INFO       **************************************************************************************
                      ** TEST                          PASS/FAIL  SIM TIME(NS)  REAL TIME(S)  RATIO(NS/S) **
                      **************************************************************************************
                      ** argon_fpga_tb.run_test_001    FAIL        17540.00          0.45       38853.11  **
                      **************************************************************************************

17540.00ns INFO       **************************************************************************************
                      **                                 ERRORS : 1                                         **
                      **************************************************************************************
                      **                               SIM TIME : 17540.00 NS                               **
                      **                              REAL TIME : 0.46 S                                    **
                      **                        SIM / REAL TIME : 37845.52 NS/S                             **
                      **************************************************************************************
```

Rysunek 5.5. Przykładowy test zakończony błędnym wynikiem w CocoTB

5.2. Testy realizowane w sprzęcie

W celu weryfikacji zaimplementowanego koprocesora przygotowano prosty program na HPS. Program sprawdza *ID* koprocesora oraz pobiera adres bufora wymiany danych. Następnie tablicę `uint64_t` o rozmiarze 128 wypełnia pseudolosowymi danymi. Standard C gwarantuje tylko $2^{15}-1$ jako minimalną wartość stałej `RAND_MAX`, opisującej maksymalną wartość zwracaną przez funkcję `rand`. Dlatego zaimplementowano generowanie pseudolosowej liczby typu `uint32_t` w sposób przedstawiony na listingu 5.1, poprzez wycinanie z wyników losowań najmłodszych bajtów, a następnie przesuwanie ich w lewo na kolejne pozycje. W analogiczny sposób zaimplementowano otrzymywanie liczb `uint64_t` jako: `( ((uint64_t)get_random32()) << 32 ) | ( (uint64_t)get_random32() )`, którymi wypełniana jest tablica z blokiem początkowym.

```
uint32_t get_random32()
{
    uint32_t x;
    x  =  rand() & 0xff;
    x |= (rand() & 0xff) << 8;
    x |= (rand() & 0xff) << 16;
    x |= (rand() & 0xff) << 24;

    return x;
}
```

Listing 5.1. Funkcja generująca pseudolosową liczbę `uint32_t` niezależnie od implementacji standardu C.

Po przygotowaniu bloku testowego, wywoływana jest funkcja `HexDump`, drukująca zawartość bloku w postaci heksadecymalnej, w sposób przedstawiony na rys.5.6.

Program kopiuje blok do bufora wymiany danych, a następnie inicjalizuje koprocesor. W tym celu kolejno:

- restartuje koprocesor, zapisując wartość 2 w rejestrze kontrolnym,
- odczytuje rejestr statusu i sprawdza czy jego wartość wynosi ′0′,
- sprawdza czy wartość rejestru kontrolnego została przestawiona na 0 po resecie,

```
OrigIn:
51 FF 4A EC 67 C6 69 73  F2 FB E3 46 29 CD BA AB
1B E8 E7 8D 7C C2 54 F8  33 9F C9 9A 76 5A 2E 63
31 58 A3 5A 66 32 0D B7  58 E9 5E D4 25 5D 05 17
9B B4 54 11 AB B2 CD C6  21 3D DC 87 0E 82 74 41
41 E1 FC 67 70 E9 3E A1  EA DC 6B 96 3E 01 7E 97
EC B0 3B FB 8F 38 5C 2A  EC 18 DB 5C 32 AF 3C 54
FB FA AA 3A 02 1A FE 43  05 3C 7C 94 FB 29 D1 E6
```

Rysunek 5.6. Początek bloku testowego wydrukowany przez funkcję `HexDump`.

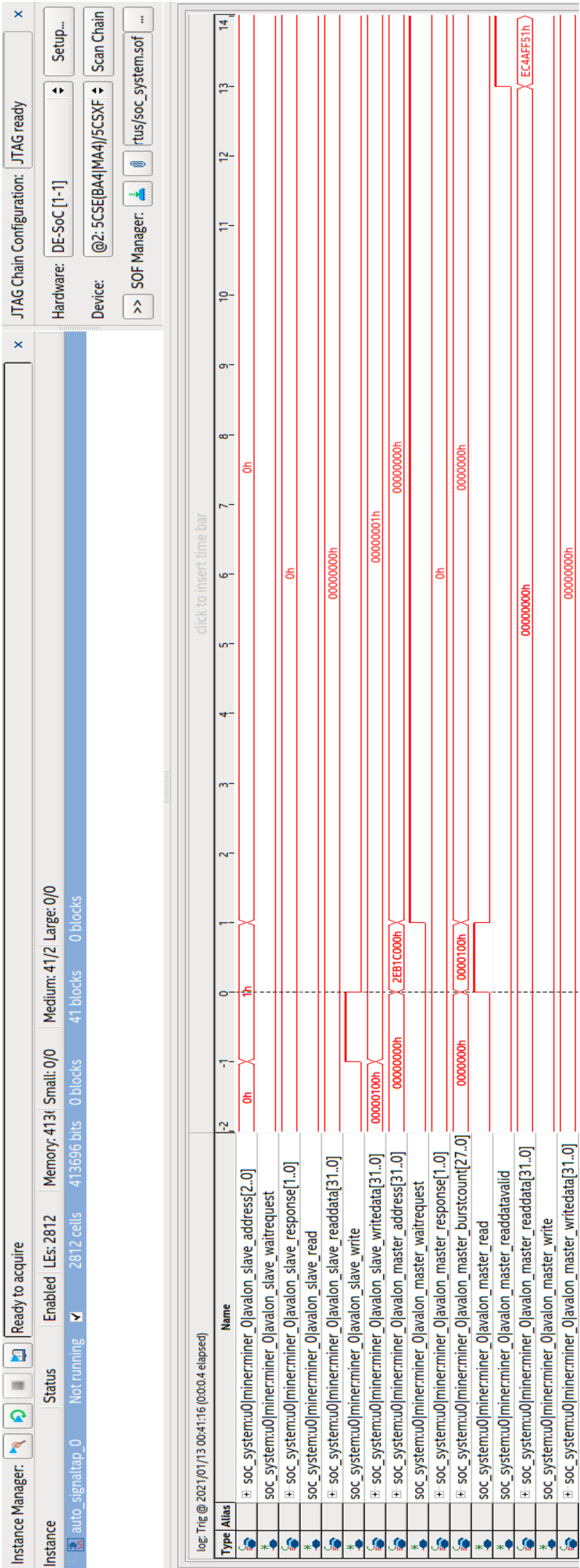
- zapisuje adres fizyczny początku bufora wymiany danych oraz długość bloku wynoszącą 256,
- uruchamia koprocesor.

W czasie kiedy koprocesor równolegle liczy, na HPS wykonywany jest zestaw operacji skopiowany z *modelu C* (listing 3.1). Po zakończeniu obliczeń na HPS i odczytaniu z rejestru statusu koprocesora informacji o tym, że koprocesor również skończył, aplikacja przechodzi do sprawdzania wyników. Najpierw wywoływana jest dwukrotnie funkcja `HexDump` drukująca kolejno referencyjny blok wynikowy obliczony na HPS oraz blok wynikowy obliczony przez koprocesor. Następnie w pętli porównywane są kolejne słowa `uint64_t` pochodzące z obydwu bloków oraz są drukowane w 2 kolumnach przedzielonych znakiem == lub `!=` zależnie od wyniku. W przypadku wystąpienia błędu drukowany jest dodatkowo numer słowa dla którego wystąpiła rozbieżność jak na rys.5.7. Dalsze sprawdzanie zostaje przerwane. Na koniec drukowany jest komunikat z wynikiem testu: `"TEST PASSED"` lub `"TEST_FAILED"`, urządzenie jest zamykane, a program się kończy wywołaniem: `exit(EXIT_SUCCESS)`.

```
7369C667EC4AFF51 == 7369C667EC4AFF51
ABBACD2946E3FBF2 == ABBACD2946E3FBF2
F854C27C8DE7E81B == F854C27C8DE7E81B
632E5A769AC99F33 == 632E5A769AC99F33
B70D32665AA35831 == B70D32665AA35831
17055D25D45EE958 == 17055D25D45EE958
C6CDB2AB1154B49B == C6CDB2AB1154B49B
4174820E87DC3D21 == 4174820E87DC3D21
i = 8
A13EE97067FCE141 != 7369C667EC4AFF51
TEST FAILED
```

Rysunek 5.7. Początek bloku testowego wydrukowany przez funkcję `HexDump`.

Dodatkowo przeprowadzono test umożliwiający ręczną weryfikację programu testującego. Przy pomocy narzędzia *SignalTap* zarejestrowano transakcje pomiędzy HPS a koprocesorem. Następnie porównano zaobserwowane, przesyłane dane z wartościami raportowanymi przez program testujący oraz sterownik. Na rys.5.8 zaprezentowano przykładowy zrzut ekranu, na którym widać moment uruchomienia koprocesora przez HPS, a następnie zażądanie odczytu danych z bufora DMA przez koprocesor.



Rysunek 5.8. Moment uruchomienia koprocesora zaobserwowany przy pomocy *SignalTap*

6. Ocena rozwiązania

6.1. Profilowanie algorytmu

Dokonano pomiarów na 2 sposoby:

1. przy pomocy programu *gprof*,
2. korzystając z funkcji `gettimeofday` w języku C.

Otrzymane wyniki znacznie się różniły, zależnie od zastosowanej metody pomiaru. Po dokonaniu analizy wyników i kodu aplikacji oraz obejrzeniu danych zarejestrowanych analizatorem logicznym z narzędzia *SignalTap*, znaleziono przyczynę, skutkującą innym niż zamierzone działaniem aplikacji. Dokładniejszy opis tamtych pomiarów oraz błąd w implementacji i sposób jego naprawy zamieszczono w załączniku 1.

Po usunięciu błędu powtórzono pomiary. Wyniki uzyskane programem *gprof* dla 2 skrajnych wartości parametru `tcost` przedstawiono w tab.6.1 i 6.2,

Tabela 6.1. Porównanie różnych implementacji, parametr `tcost`: 1, liczba powtórzeń: 10000, liczba wywołań `FillBlock`: 2320000, `blake2b_compress`: 2590000.

	ARM (referencyjna)	ARM + FPGA	ARM + Neon	ARM + FPGA + Neon
1 wywołanie `FillBlock`	27,18us	18,30us	30,95us	18,47us
% czasu spędzony w `FillBlock`	59,33%	54,96%	67,07%	55,20%
1 wywołanie `blake2b_compress`	11,98us	12,08us	11,98us	11,98us
% czasu spędzony w `blake2b_compress`	29,20%	40,52%	28,99%	39,99%

Tabela 6.2. Porównanie różnych implementacji, parametr `tcost`: 10, liczba powtórzeń: 10000, liczba wywołań `FillBlock`: 23920000, `blake2b_compress`: 2590000.

	ARM (referencyjna)	ARM + FPGA	ARM + Neon	ARM + FPGA + Neon
1 wywołanie `FillBlock`	26,64us	18,64us	31,66us	15,88us
% czasu spędzony w `FillBlock`	81,55%	89,59%	93,47%	89,57%
1 wywołanie `blake2b_compress`	12,15us	11,63us	11,95us	11,90us
% czasu spędzony w `blake2b_compress`	4,06%	6,05%	3,82%	5,98%

Wszystkie implementacje posiadają taki sam kod funkcji `blake2b_compress`. Dla każdego przypadku funkcja `blake2b_compress` wykorzystywana jest tylko podczas ini-

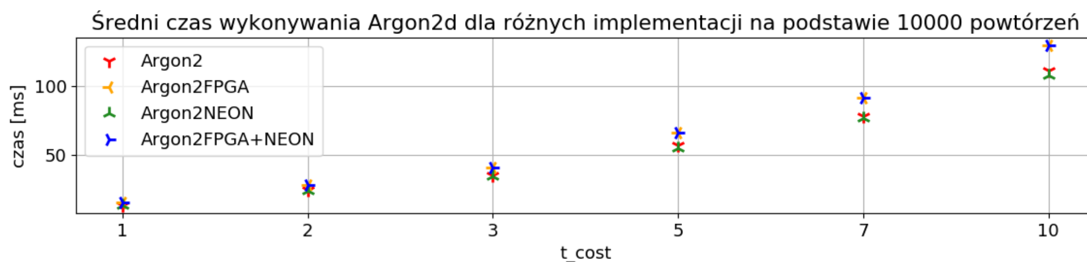
cjalizacji bloku, dlatego liczba jej wywołań nie zależy od wartości parametru `tcost`, a czasy wykonywania tej funkcji są zbliżone dla wszystkich przypadków. Funkcja ta nie jest wywoływana w pobliżu żadnego spośród akcelerowanych fragmentów. W związku z tym dane, z których ona korzysta, powinny być w każdym przypadku ułożone w pamięci w taki sam sposób, a różnice czasów powinny mieć w głównej mierze charakter losowy.

Mimo zastosowania podczas kompilacji flag `-O3 -mfpu=neon` ustawiających najwyższy poziom optymalizacji, nie stwierdzono w wygenerowanym kodzie maszynowym obecności instrukcji SIMD dla jednostki *Neon* w miejscach innych niż te, gdzie w kodzie C je umieszczono explicite.

Drugim pomiarem był pomiar czasu z wykorzystaniem funkcji `gettimeofday`. Funkcja ta umożliwia odczyt godziny z zegara, z dokładnością do 1ms. W ramach pomiaru kolejno:

1. generowano pseudolosowy zestaw danych testowych,
2. pobierano czas rozpoczęcia pomiaru,
3. uruchamiano w pętli `for(unsigned int i = N; i--; )` N-krotnie cały algorytm,
4. pobierano czas zakończenia pomiaru,
5. różnicę czasów zapisywano w logu.

Wyniki uzyskane przy pomocy tej funkcji zamieszczono w tab.6.3 oraz na rys.6.1.



Rysunek 6.1. Rzeczywisty czas prowadzenia obliczeń, dla poprawionej implementacji, w funkcji parametru `tcost`, uśredniony na podstawie 10000 powtórzeń

Tabela 6.3. Średni czas dla różnych implementacji Argon2d na podstawie 10000 powtórzeń. Pomiar z usuniętym wywołaniem funkcji `usleep`.

Wartość `tcost`	ARM (ref)	ARM + FPGA	ARM + Neon	ARM + FPGA + Neon
1	13,57 ms	15,41 ms	13,39 ms	15,36 ms
2	24,52 ms	28,22 ms	24,06 ms	27,87 ms
3	35,09 ms	40,78 ms	34,18 ms	40,71 ms
5	56,75 ms	66,24 ms	55,20 ms	66,08 ms
7	78,30 ms	91,57 ms	76,56 ms	91,58 ms
10	110,97 ms	129,37 ms	107,44 ms	129,18 ms

Zgodnie z przypuszczeniami w powtórzonych testach, większość wyników uzyskanych programem *gprof* nie uległa znaczącej zmianie. Największe zmiany można zaobserwować

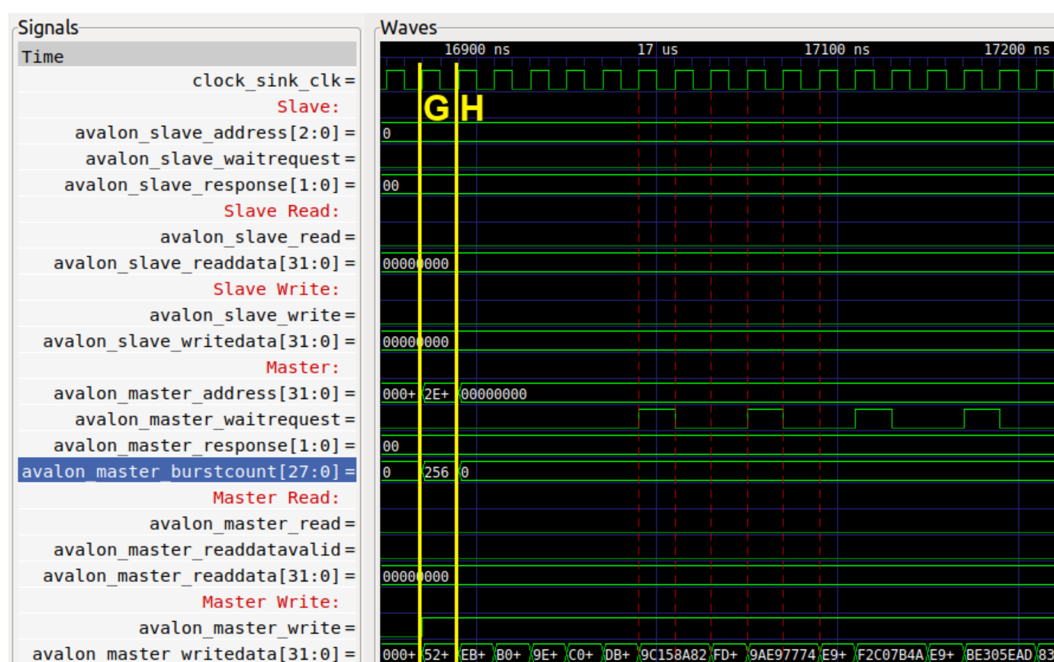
w tab.6.2, dla implementacji wykorzystujących koprocesor. Wynikają one z faktu usunięcia wywołania funkcji `usleep(1)`, co powoduje że w trakcie wykonywania funkcji `FillBlock` proces jest częściej w stanie pracującym, a co za tym idzie profiler zapisuje również więcej próbek z tej funkcji.

Porównując natomiast wyniki z tab.Z1.3 i tab.6.3 uzyskane poprzez bezpośredni pomiar czasu rozpoczęcia i zakończenia obliczeń, można zauważyć prawie dwukrotne skrócenie czasu obliczeń prowadzonych z wykorzystaniem koprocesora w części programowalnej, spowodowane uwzględnieniem w implementacji sposobu zarządzania procesami przez system operacyjny. Tak jak wcześniej, zastosowanie jednostki obliczeniowej *Neon* przyniosło nieznaczną poprawę wyniku. Implementacja na HPS działa około 10-15% szybciej niż korzystająca z koprocesora.

6.2. Pomiar czasu obliczania funkcji G przez koprocesor

Przeprowadzono pomiary czasu obliczeń realizowanych przez koprocesor. W tym celu przygotowano oddzielny program, który pobierał jako argument liczbę N powtórzeń, generował pseudolosowy zbiór danych i inicjalizował koprocesor. Następnie uruchamiał procedurę testu dla implementacji na tylko i wyłącznie HPS, a potem na HPS z wykorzystaniem koprocesora. Pomiaru czasu dokonywano wyznaczenie różnicy między czasem końca, a początku pętli, wewnątrz której N-krotnie obliczano funkcję G.
Dla $N = 10^8$ powtórzeń uzyskano średnio:

- 24,82us dla realizacji na samym HPS,
- 29,37us dla realizacji wykorzystującej koprocesor.

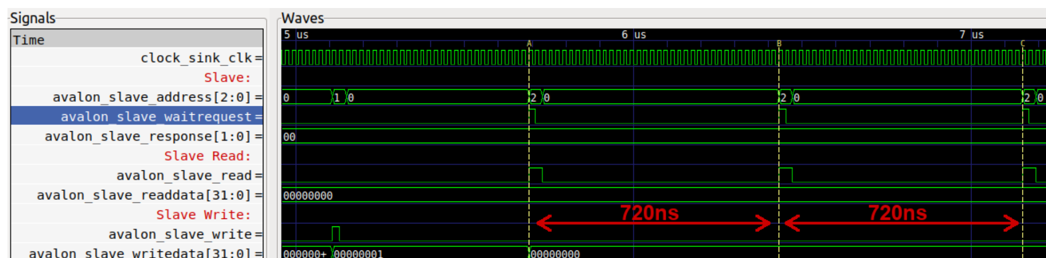


Rysunek 6.2. Koniec transakcji

Przy pomocy narzędzia *SignalTap* zarejestrowano komunikację na magistrali Avalon w trakcie cyklu pracy koprocesora. W chwili czasowej:

- $t_{start} = 5,11$us zapisywana do rejestru kontrolnego koprocesora jest wartość 1, uruchamiająca koprocesor (rys.6.4 A-C),
- $t = 5,40$us pamięć RAM zaczyna wysyłać dane do koprocesora (rys.6.4 D),
- $t = 5,95$us HPS po raz pierwszy odczytuje rejestr *status* koprocesora (rys.6.4 E-F),
- $t = 16,89$us koprocesor rozpoczyna zapis wyników do bufora w RAM (rys.6.2 G-H),
- $t_{finish} = 24,47$us koprocesor kończy pracę, takt po tym jak HPS sprawdził rejestr *status*,
- $t_{end} = 25,17$us HPS odczytuje, że koprocesor zakończył pracę,
- $t_{reset} = 33,17$us HPS potwierdza przeniesienie danych z bufora w pamięci RAM poprzez zapisanie wartości 0 do rejestru kontrolnego, jednocześnie powodując reset koprocesora.

Procesor sprawdzał stan koprocesora co 0,72us, aktywnie oczekując w pętli:
`while(!ioctl(fminer, MINER_IOC_READ_STATUS, NULL)){}`.



Rysunek 6.3. Okresowe sprawdzanie rejestru *Status* przez HPS

Zmierzony odczyt 1KB danych z bufora w pamięci RAM przez HPS, a następnie zapis do rejestru kontrolnego koprocesora zajmuje 8us. Z kolei zapis z HPS do bufora oraz przestawienie rejestru w koprocesorze zajmują 1us. Można przypuszczać, że głównym źródłem różnicy jest fakt xorowania wyników z danymi w pamięci podręcznej procesora podczas odczytu danych z bufora.

Zmierzony czas wymiany danych pomiędzy koprocesorem a buforem DMA, niezależnie od kierunku wynosi: $\frac{3}{2} \cdot 256/50MHz = 7,68$us, gdzie 256 to liczba przesyłanych słów, $50MHz$ to częstotliwość zegara koprocesora, a czynnik na początku wyrażenia odpowiada 'współczynnikowi wypełnienia transakcji'. Przerwy w transakcji najprawdopodobniej wynikają z niedopracowanej konfiguracji połączenia: RAM $\longleftrightarrow$ *AXI* $\longleftrightarrow$ *Avalon* $\longleftrightarrow$ koprocesor, na co wskazuje ich powtarzalny charakter. Wyróżniono przykładowy fragment transferu na rys.6.2, gdzie przy pomocy przerywanych linii zaznaczono kolejne zbocza narastające zegara. Można zauważyć że sygnał *_waitrequest* jest co trzeci takt podnoszony na czas trwania 1 taktu.

Rysunek 6.4. Początek transakcji

Długość pojedynczego cyklu pracy bloku *Mix* koprocesora jest deterministyczna podczas drugiej rundy i wynosi 21 taktów zegara. Pomiędzy zakończeniem odczytu, a rozpoczęciem zapisu danych, koprocesor wykonuje 8 takich cykli. W związku z tym, przerwa pomiędzy 2 transakcjami wynosi około 3,4us.

Całkowity czas potrzebny na przetworzenie 1 bloku z wykorzystaniem koprocesora można zatem oszacować jako:
$1us + 2 \cdot 7,7us + 3,4us + 8us = 26,8us$. Jest to porównywalny czas do osiąganego przy prowadzeniu obliczeń tylko i wyłącznie na HPS.

Głównymi ograniczeniami są, zgodnie z przewidywaniami, czasy transferu danych. Gdyby dalej wymieniać dane przez bufor w pamięci RAM, to można by maksymalnie 12-krotnie skrócić czas wymiany danych między RAM a koprocesorem. Pamięć dostępna na wykorzystywanej płycie umożliwia wymianę danych o szerokości 32 bitów z częstotliwością do 400MHz. W bloku *Comm* koprocesora można by wtedy umieścić serializator/deserializator umożliwiający ze strony magistrali wymianę słów o szerokości 32 bitów z częstotliwością 400MHz, a ze strony reszty koprocesora, np. 64 bitów przy 200MHz. Dodatkowo należało by wnikliwie przeanalizować strukturę połączenia między RAM, a koprocesorem i spróbować wyeliminować przyczynę przerw w transferach danych. Gdyby się udało wprowadzić obydwie zmiany, czas transferu można by skrócić z 14,4us do nawet 1,2us. Stanowiło by to skrócenie czasu cyklu pracy koprocesora o około połowę. Dodatkowo zwiększenie częstotliwości pracy koprocesora umożliwi 2- lub 4-krotne skrócenie drugiej rundy mieszania bloku, co się przełoży na dodatkowe 2-3us, czyli kolejne 8-12% zysku. Alternatywnym podejściem była by implementacja obsługi portu ACP. Wtedy koprocesor łączył by się bezpośrednio z pamięcią podręczną L2 procesora, zmniejszając liczbę wszystkich transferów danych przypadających na cykl pracy koprocesora o połowę.

6.3. Zużycie zasobów

Na rys.6.5 przedstawiono fragment raportu wygenerowanego w środowisku *Quartus* po zakończeniu kompilacji i syntezy układu. W tab.6.4 przedstawiono dokładniejsze wyniki uwzględniające zużycie zasobów przez poszególne bloki lub grupy bloków.

Tabela 6.4. Zużycie zasobów w układzie programowalnym z wyszczególnieniem bloków.

	Bloki *ALM* [46]	*ALUT*	Rejestry logiczne	Pamięć blokowa [b]	Pamięci *M10K*	Bloki DSP
Całkowite (%)	4513 (28%)	-	3228 (-)	8192 (< 1%)	-	39 (46%)
Miner	3207,1	5314	1540	8192	7	39
Comm	153,1	235	146	0	0	0
Reg	309,2	443	121	8192	7	0
Mix	2744,8	4636	1273	0	0	39
Interconnecty	1185,4	1649	1353	0	0	0
Inne	2,5	1	3	0	0	0

Zaprojektowany koprocesor bez problemu mieści się w układzie programowalnym, zajmując przy tym mniej niż połowę jego powierzchni. Gdyby po poprawieniu przepustowości połączenia *RAM* ⟷ *koprocesor* blok *Mix* okazał się ograniczać wydajność koprocesora, możliwym wydaje się skrócenie czasu jego pracy o prawie połowę poprzez realizację obydwu funkcji `fBlaMka(x,y)` oraz `xorrot(x,y,n)` w trakcie trwania 1 taktu. Wiązałoby się ze wzrostem zużycia zasobów przez blok *Mix*. Gdyby takiej potrzeby nie było, to ze sporym prawdopodobieństwem możliwe by było nawet zaprogramowanie w układzie dwóch takich koprocesorów.

Największym ograniczeniem jest liczba dostępnych bloków DSP, które są wykorzystywane w bloku *Mix*. Wykorzystanie 39 spośród 84 dostępnych oznacza, że przy próbie umieszczenia 2 koprocesorów w części reprogramowalnej pozostanie tylko 6 wolnych bloków, co z kolei podniesie trudność efektywnej implementacji, a dokładniej trasowania połączeń między poszczególnymi blokami. Wydłużenie połączeń z kolei może prowadzić do ograniczenia maksymalnej częstotliwości z jaką może pracować układ, obniżając jednocześnie efektywność każdego z koprocesorów. Możliwą wydaje się redukcja liczby potrzebnych bloków DSP poprzez odpowiednią dekompozycję funkcji `fBlaMka(x,y)` i zastosowanie metod arytmetyki rozproszonej. Skutkowało by to wzrostem wykorzystania bloków pamięci *M10K*, ale nie powinno to być problemem, ponieważ obecnie koprocesor wykorzystuje <1% dostępnych bloków *M10K*.
Dodatkowo, dołożenie drugiego koprocesora oznacza mniejszy niż dwukrotny wzrost zużycia bloków *ALM*. Wynika to z faktu, że mogły by współdzielić znaczną część logiki *interconnectu 0*, zaznaczonego na rys.Z2.1 kolorem jasnozielonym, łączącego je z *HPS*.

Drugi koprocesor mógłby realizować funkcję `blake2b_compress`, która była drugą najdłużej trwającą w trakcie wykonywania algorytmu, gdyż jest bardzo podobna do obecnie realizowanej w koprocesorze funkcji. `FillBlock`. Różnice w głównej mierze by się wiązały z modyfikacją części operandów w bloku *Mix*.
Alternatywnie, można zaimplementować algorytm w sposób umożliwiający wykorzystanie obydwu koprocesorów równocześnie w sytuacji gdy parametr `tcost` >= 2.

Flow Status	Successful - Wed Jan 13 19:40:14 2021
Quartus Prime Version	18.1.0 Build 625 09/12/2018 SJ Lite Edition
Revision Name	soc_system
Top-level Entity Name	ghrd
Family	Cyclone V
Device	5CSEMA4U23C6
Timing Models	Final
Logic utilization (in ALMs)	4,513 / 15,880 (28 %)
Total registers	3228
Total pins	230 / 314 (73 %)
Total virtual pins	0
Total block memory bits	8,192 / 2,764,800 (< 1 %)
Total DSP Blocks	39 / 84 (46 %)
Total HSSI RX PCSs	0
Total HSSI PMA RX Deserializers	0
Total HSSI TX PCSs	0
Total HSSI PMA TX Serializers	0
Total PLLs	0 / 5 (0 %)
Total DLLs	1 / 4 (25 %)

Rysunek 6.5. Zużycie zasobów sprzętowych przez koprocesor i połączenie do *HPS*

6.4. Porównanie z realizacją ASIC

J. Rossetti oraz W. Ruggiero zaproponowali implementację pojedynczej rundy funkcji mieszającej G dla układu ASIC wykonanym w technologii 90nm [22]. W roli implementacji odniesienia wykorzystali implementację funkcji Blake2b, której modyfikacją jest runda funkcji G. W tabeli 6.5 przedstawiono uzyskane przez nich wyniki dla kolejno:

- implementacji referencyjnej *blake2b-ref,*
- najlepszej pod względem wypośrodkowania między przepustowością, powierzchnią, a poborem mocy implementacji *seq-blamka-default,*
- implementacji o najwyższej przepustowości *comb-blamka-default.*

Tabela 6.5. Wyniki dla implementacji rundy funkcji G dla układu ASIC w technologii 90nm.

Nazwa	Max f zegara	Latencja	Przepustowość
blake2b-ref	225,734 MHz	26,58 ns	9,631 Gbit/s
seq-blamka-default	36,550 MHz	164,16ns	1,559 Gbit/s
comb-blamka-default	-	111,52 ns	2,296 Gbit/s

W pracy zaproponowano implementację dla układu FPGA realizującą od razu 2 rundy funkcji G, w postaci bloku *Mix*. Implementacji dla układu ASIC odpowiada 1/4 bloku *Mix*, ponieważ blok ten oblicza pojedynczą rundę na 4 niezależnych wektorach o rozmiarze

256 bitów. Maksymalna częstotliwość zegara wynosi 131MHz, przy czym sprawdzenie czy to blok *Mix* ogranicza częstotliwość zegarów wymagałoby dalszych testów. Przeprowadzenie obliczeń zajmuje 20 taktów. W tym czasie przetworzone dwukrotnie zostają dane o długości 256 bitów. Daje to średnią przepustowość na poziomie 3.123 Gbit/s, w przeliczeniu na rundę, czyli wzrost o 36% względem implementacji ASIC o najwyższej przepustowości. Kolejną zaletą rozwiązania na FPGA jest fakt, że bloki wewnątrz układu można dostosować do przyszłych zmian algorytmu. Do wad natomiast należy zaliczyć zdecydowanie większą powierzchnię oraz niemal pewny wyższy pobór mocy.

6.5. Dalsze prace

Przede wszystkim potrzeba rozszerzyć ocenę rozwiązania o pomiary pobieranej mocy. O ile w przypadku potrzeby szybkiego łamania haseł, kluczowym kryterium jest przepustowość, o tyle w przypadku zastosowania systemu w koparkach kryptowalut ważniejszym parametrem jest stosunek przepustowości do pobieranej mocy.

Należałoby spróbować dokonać dalszych optymalizacji bloku *Mix*. Potencjalnie można by zredukować zużycie zasobów sprzętowych poprzez podzielenie fragmentów mieszanego wiersza lub kolumny na pary, gdzie w obrębie każdej pary 1 wiersz byłby opóźniony o takt, dzięki czemu 2 oddzielne wektory mogłyby współdzielić bloki realizujące funkcje `fBlaMka` i `xor_rotr64`. Z kolei w celu próby dalszego skrócenia obliczeń, należałoby rozważyć scalenie funkcji `fBlaMka` oraz `xor_rotr64`, co by pozwoliło skrócić liczbę taktów potrzebnych na obliczenia o prawie połowę, kosztem poszerzenia szyn danych.

Największy zysk szybkości przetwarzania danych może wynikać z poprawy mechanizmów wymiany danych z procesorem. Można próbować wprowadzać szereg zmian:

- ustalić i wyeliminować przyczynę przerw transmisji danych przez *Interconnect*,
- wprowadzić dodatkową domenę zegarową, żeby magistrala była taktowana z jak najwyższą częstotliwością (pamięć obsługuje maksymalnie 400MHz),
- zwiększyć szerokość szyny danych na magistrali do 64 albo 128 bitów,
- stworzyć połączenie przez most F2S zamiast F2H (rys. 3.3)
- zastąpić wymianę danych przez bufor w RAM, wymianą bezpośrednio z pamięcią podręczną L2 procesora poprzez port ACP.

Warto rozszerzyć system o obsługę przerwań wysyłanych przez koprocesor.

Na bazie projektu jest rozwijany publicznie dostępny projekt demonstracyjny *DE0-SoC GSRD* (*Golden System Reference Design*) [47], oferujący minimalny, działający zestaw komponentów potrzebnych, aby można było szybko prototypować następne realizacje sprzętowo-programowe na płycie *Terasic DE0-Nano-SoC*. Wyróżnia go na tle podobnych 'projektów-szablonów' [48] wsparcie dla zestawu narzędzi *Buildroot*, przez co użytkownik dostaje gotowe, prekonfigurowane narzędzie umożliwiające zbudowanie systemu Linux dostosowanego do indywidualnych potrzeb konkretnego projektu. Dodatkowo ułatwiona została weryfikacja części sprzętowej, poprzez przygotowanie konfiguracji dla otwartoźró-

dłowych: frameworku *CocoTB* oraz symulatora *GHDL*, wraz z prekompilacją szyfrowanych modułów od dostawcy narzędzia *Quartus*. W miarę rozwoju projektu, cennym dodatkiem mogło by być dodanie wsparcia dla kosymulacji sprzętu i programów z wykorzystaniem otwartoźródłowego symulatora QEMU oraz dodanie tzw. ciągłej integracji, czyli stworzenie zautomatyzowanej procedury testowania projektów sprzętowo-programowych umożliwiającej weryfikację zmian z repozytorium najpierw w symulacji, a na koniec bezpośrednio na płytach podłączonych do sieci.

7. Podsumowanie

W pracy przedstawiono koncepcję i implementację realizacji sprzętowo-programowej algorytmu Argon2d z wykorzystaniem układu SoC FPGA. Zaimplementowano mechanizm wymiany danych pomiędzy częścią programową a sprzętową. Zaprojektowano opisany w języku *VHDL*, realizujący wskazany na podstawie profilowania realizacji programowej fragment algorytmu, koprocesor obsługujący wymianę danych poprzez interfejsy Avalon-MM. Dostosowano funkcję mieszania do architektury części programowalnej układu. Przygotowano obraz systemu operacyjnego dla wykorzystywanej płyty oraz napisano dla niego sterownik do koprocesora. Zademonstrowano wykorzystanie funkcji realizowanej sprzętowo w aplikacji użytkownika. Przygotowano dodatkowe warianty implementacji programowej oraz sprzętowo-programowej wykorzystujące rozszerzenie *Neon* architektury ARM w celu dalszej akceleracji obliczeń.

Pokazano sposoby weryfikacji projektu na etapie symulacji oparte na otwartych narzędziach. Wskazano możliwości rozwoju w kierunku stworzenia systemu automatycznego procesu testowania projektu. Zwrócono uwagę na nieoczywiste błędy mogące występować przy podobnych pomiarach. Dodatkowo zgłoszono i naprawiono kilka błędów w wykorzystywanych bibliotekach.

Dokonano pomiarów i analizy czasu wykonania algorytmu oraz porównania wyników uzyskanych dla akcelerowanego fragmentu z opublikowanymi dla realizacji z wykorzystaniem układu ASIC. Nie udało się uzyskać przyspieszenia obliczeń w porównaniu z implementacją czysto programową. Wskazano natomiast szereg zmian, które należałoby wprowadzić w projekt prototypu, w celu podniesienia wydajności rozwiązania, potencjalnie skutkujących kilkukrotnym skróceniem czasu prowadzenia obliczeń.

Na bazie przedstawionego projektu powstaje szablon o otwartych źródłach, umożliwiający szybkie prototypowanie zbliżonych rozwiązań dla wykorzystywanej platformy. W związku z przyjętym na początku prac założeniem o dążeniu do maksymalizacji przenaszalności rozwiązania, projekt można będzie niewielkim nakładem pracy przenosić na inne platformy SoC FPGA firmy Intel. A przeniesienie na platformy wykorzystujące układy SoC FPGA innych producentów będzie wymagało jedynie skonfigurowania na nowo połączenia między HPS, pamięcią RAM i układem FPGA w obrębie systemu oraz ewentualnie zmiany interfejsów komunikacyjnych koprocesora.

Projekt ten należy traktować raczej jako demonstrator sprawdzający możliwość wykorzystania układów SoC FPGA do akceleracji algorytmu projektowanego z myślą o utrudnieniu akceleracji sprzętowej. Uzyskane wyniki potwierdzają trudność uzyskania skutecznej akceleracji algorytmu. Zastosowanie zaproponowanych w rozdziale 6.5 zmian w projekcie powinno zwiększyć wydajność akceleracji.

Bibliografia

[1] R. L. Rivest, “The MD5 Message-Digest Algorithm”, RFC Editor, kw. 1992. adr.: http://www.rfc-editor.org/rfc/rfc1321.txt.

[2] S. Turner i L. Chen, “Updated Security Considerations for the MD5 Message-Digest and the HMAC-MD5 Algorithms”, RFC Editor, mar. 2011. adr.: http://www.rfc-editor.org/rfc/rfc6151.txt.

[3] *Git - Git Objects*, ostatni dostęp 29.01.2021. adr.: https://git-scm.com/book/en/v2/Git-Internals-Git-Objects.

[4] *cryptography - Why doesn't Git use more modern SHA? - Stack Overflow*, ostatni dostęp 29.01.2021. adr.: https://stackoverflow.com/questions/28159071/why-doesnt-git-use-more-modern-sha/47838703#47838703.

[5] S. Whited, “Best practices for password hashing and storage”, IETF Secretariat, kw. 2020. adr.: http://www.ietf.org/internet-drafts/draft-whited-kitten-password-storage-02.txt.

[6] *Password Hashing Competition*, ostatni dostęp 29.01.2021. adr.: https://www.password-hashing.net/.

[7] A. Biryukov, D. Dinu i D. Khovratovich, “Argon2: the memory-hard function for password hashing and other applications”, mar. 2017, ostatni dostęp 29.01.2021. adr.: https://www.cryptolux.org/index.php/Argon2.

[8] S. Sayeed i H. Marco-Gisbert, “Assessing Blockchain Consensus and Security Mechanisms against the 51% Attack”, *Applied Sciences*, t. 9, s. 1788, 9 kw. 2019, ISSN: 2076-3417. DOI: 10.3390/app9091788. adr.: https://www.mdpi.com/2076-3417/9/9/1788.

[9] S. Nakamoto, “Bitcoin: A Peer-to-Peer Electronic Cash System”, 2018, ostatni dostęp 24.01.2021. adr.: www.bitcoin.org.

[10] A. R. Zamanov, V. A. Erokhin i P. S. Fedotov, “ASIC-resistant hash functions”, t. 2018-January, Institute of Electrical i Electronics Engineers Inc., mar. 2018, s. 394–396, ISBN: 9781538643396. DOI: 10.1109/EIConRus.2018.8317115.

[11] *Altera's shipping its first SoC FPGAs | EE Times*, ostatni dostęp 29.01.2021. adr.: https://www.eetimes.com/alteras-shipping-its-first-soc-fpgas/.

[12] *Xilinx Ships First Zynq-7000 Devices, the World's First Extensible Processing Platform*, ostatni dostęp 29.01.2021. adr.: https://www.design-reuse.com/news/27981/xilinx-zynq-7000-shipping.html.

[13] *Xilinx Introduces Zynq-7000 Family, Industry's First Extensible Processing Platform*, ostatni dostęp 29.01.2021. adr.: https://www.prnewswire.com/news-releases/xilinx-introduces-zynq-7000-family-industrys-first-extensible-processing-platform-117132003.html.

[14] A. Biryukov, D. Dinu i D. Khovratovich, “Argon2: New generation of memory-hard functions for password hashing and other applications”, Institute of Electrical i

Electronics Engineers Inc., maj 2016, s. 292–302, ISBN: 9781509017515. DOI: 10.1109/EuroSP.2016.31.

[15] *khovratovich/Argon2: Memory-hard scheme Argon2*, ostatni dostęp 29.01.2021. adr.: https://github.com/khovratovich/Argon2.

[16] M.-J. Saarinen i J.-P. Aumasson, "The BLAKE2 Cryptographic Hash and Message Authentication Code (MAC)", RFC Editor, RFC 7693, list. 2015.

[17] R. Nandakumar i A. C. Kumar, "ARGON2id IP Core", *International Research Journal of Engineering and Technology*, t. 06, s. 3086–3092, 06 2019, ostatni dostęp 15.01.2021, ISSN: 2395-0072. adr.: www.irjet.net.

[18] *phc-discussions - Re: [PHC] Argon2 CPU/GPU benchmarks*, ostatni dostęp 29.01.2021. adr.: https://lists.openwall.net/phc-discussions/2015/10/15/1.

[19] *[PHC] Argon2 CPU/GPU benchmarks*, ostatni dostęp 29.01.2021. adr.: https://discussions.password-hashing.narkive.com/pYOIcGhK/phc-argon2-cpu-gpu-benchmarks.

[20] *John: doc/NEWS | Fossies*, ostatni dostęp 29.01.2021. adr.: https://fossies.org/linux/john/doc/NEWS.

[21] *Ondrej Mosnáček / argon2-gpu · GitLab*, ostatni dostęp 29.01.2021. adr.: https://gitlab.com/omos/argon2-gpu.

[22] J. F. Rossetti i W. V. Ruggiero, "Hardware implementation for permutation function of multiplication-hardened sponge BlaMka", Institute of Electrical i Electronics Engineers Inc., czer. 2017, ISBN: 9781509058594. DOI: 10.1109/LASCAS.2017.7948054.

[23] S. Atiwa, Y. Dawji, A. Refaey i S. Magierowski, "Accelerated Hardware Implementation of BLAKE2 Cryptographic Hash for Blockchain", Institute of Electrical i Electronics Engineers (IEEE), list. 2020, s. 1–6, ISBN: 9781728154428. DOI: 10.1109/ccece47787.2020.9255709.

[24] M. V. Beirendonck, L. C. Trudeau, P. Giard i A. Balatsoukas-Stimming, "A LYrA2 FPGA core for LyRa2Rev2-based cryptocurrencies", t. 2019-May, Institute of Electrical i Electronics Engineers Inc., 2019, ISBN: 9781728103976. DOI: 10.1109/ISCAS.2019.8702498.

[25] *ARM Compiler toolchain Compiler Reference*, ostatni dostęp 29.01.2021. adr.: https://developer.arm.com/documentation/dui0491/c/Using-NEON-Support/Logical-operations.

[26] *AN 796: Cyclone V and Arria V SoC Device Design Guidelines*, ostatni dostęp 29.01.2021. adr.: https://www.intel.com/content/www/us/en/programmable/documentation/doq1481305867183.html.

[27] "HPS SoC Boot Guide-Cyclone V SoC Development Kit", 2016, ostatni dostęp 29.01.2021. adr.: www.altera.com.

[28] B. Zabołotny, P. Araszkiewicz i M. Rawski, “Sprzętowo-programowa akceleracja algorytmu Argon2d z wykorzystaniem układu SoC FPGA”, *Przegląd Telekomunikacyjny*, s. 5–10, 2 lut. 2020. DOI: 10.15199/59.2020.2.1.

[29] *SIMD ISAs | Neon Intrinsics Reference – Arm Developer*, ostatni dostęp 29.01.2021. adr.: https://developer.arm.com/architectures/instruction-sets/simd-isas/neon/intrinsics.

[30] J. G. CTH i G. Research, “A Structured VHDL Design Method”, ostatni dostęp 30.01.2021. adr.: https://www.gaisler.com/doc/structdesign.pdf.

[31] Intel, *Avalon® Interface Specifications*, ostatni dostęp 02.01.2021, grud. 2020. adr.: https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/manual/mnl_avalon_spec.pdf.

[32] Intel, *Embedded Memory (RAM: 1-PORT, RAM: 2-PORT, ROM: 1-PORT, and ROM: 2-PORT) User Guide*, ostatni dostęp 03.01.2021, mar. 2020. adr.: https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/ug/ug_ram_rom.pdf.

[33] *DE0-Nano-SoC CD*, ostatni dostęp 29.01.2021. adr.: http://download.terasic.com/downloads/cd-rom/de0-nano-soc/.

[34] *Basics questions on System ID Peripheral - Intel Community*, ostatni dostęp 29.01.2021. adr.: https://community.intel.com/t5/Nios-II-Embedded-Design-Suite/Basics-questions-on-System-ID-Peripheral/td-p/110470.

[35] *Device Tree Usage - eLinux.org*, ostatni dostęp 29.01.2021. adr.: https://elinux.org/Device_Tree_Usage#Understanding_the_compatible_Property.

[36] *Linux Device Drivers: Tutorial for Linux Driver Development*, ostatni dostęp 29.01.2021. adr.: https://www.apriorit.com/dev-blog/195-simple-driver-for-linux-os.

[37] *Debugging by printing - eLinux.org*, ostatni dostęp 29.01.2021. adr.: https://elinux.org/Debugging_by_printing.

[38] D. Howells, P. E. McKenney, W. Deacon i P. Zijlstra, *LINUX KERNEL MEMORY BARRIERS*, ostatni dostęp 29.01.2021. adr.: https://www.kernel.org/doc/Documentation/memory-barriers.txt.

[39] B. Blinn, *Memory Barriers*, ostatni dostęp 29.01.2021, 2009. adr.: http://bruceblinn.com/linuxinfo/MemoryBarriers.html.

[40] *The Buildroot user manual*, ostatni dostęp 29.01.2021. adr.: https://buildroot.org/downloads/manual/manual.html.

[41] *fedora - How can I make a script in /etc/init.d start at boot? - Unix & Linux Stack Exchange*, ostatni dostęp 29.01.2021. adr.: https://unix.stackexchange.com/questions/20357/how-can-i-make-a-script-in-etc-init-d-start-at-boot/20360#20360.

[42] W. Zabołotny, *DE0 Nano SoC (Atlas-SoC) support for Buildroot 2018.08.2*, ostatni dostęp 29.01.2021, paź. 2018. adr.: https://groups.google.com/g/alt.sources/c/UjyhVy8xiPU/m/jJGQoIxYBgAJ.

[43] *Precompile Vendor Primitives — GHDL 0.36-dev documentation*, ostatni dostęp 29.01.2021. adr.: https://ghdl.readthedocs.io/en/stable/building/PrecompileVendorPrimitives.html.

[44] D. A. S. C. of the IEEE Computer Society, *IEEE Standard for VHDL Language Reference Manual IEEE Computer Society Developed by the Design Automation Standards Committee*. 2019, ISBN: 9781504461368. adr.: http://www.ieee.org/web/aboutus/whatis/policies/p9-26.html..

[45] B. Zabołotny, *Avalon byteenable and readonly phase exception · Issue #2309 · cocotb/cocotb*, ostatni dostęp 03.01.2021, grud. 2020. adr.: https://github.com/cocotb/cocotb/issues/2309.

[46] "Adaptive Logic Module (ALM) Definition", ostatni dostęp 29.01.2021. adr.: https://www.intel.com/content/www/us/en/programmable/quartushelp/17.0/reference/glossary/def_alm.htm.

[47] B. Zabołotny, *Bartosz M. Zabolotny / DE0-SoC GSRD · GitLab*, ostatni dostęp 29.01.2021. adr.: https://gitlab.com/bzab/de0-soc-gsrd/.

[48] *GSRD v14.0 - Compiling Golden Hardware Reference Design | Documentation | RocketBoards.org*, ostatni dostęp 27.12.2020. adr.: https://rocketboards.org/foswiki/Documentation/GSRD140CompileHardwareDesign.

Spis rysunków

Spis tabel

Spis listingów

Spis załączników

Załącznik 1. Nieudane pomiary

Pomiary dokonane programem *gprof* dla 2 skrajnych wartości parametru `tcost` zostały przedstawione w tab.Z1.1 oraz tab.Z1.2. Jednoznacznie wskazują na to, że zastąpienie funkcji `XORBlock` operacją wektorowego xor w jednostce *Neon*, spowodowało wydłużenie czasu trwania funkcji `FillBlock`.

Tabela Z1.1. Porównanie różnych implementacji, parametr `tcost`: 1, liczba powtórzeń: 10000, liczba wywołań `FillBlock`: 2320000, `blake2b_compress`: 2590000.

Funkcja \ Implementacja	ARM (ref)	ARM + FPGA	ARM + Neon	ARM + FPGA + Neon
1 wywołanie `FillBlock`	26,68us	18,27us	30,73us	19,20us
% czasu spędzony w `FillBlock`	58,70%	55,20%	67,04%	56,03%
1 wywołanie `blake2b_compress`	12,09us	11,63us	12,05us	11,75us
% czasu spędzony w `blake2b_compress`	29,69%	39,24%	29,33%	38,28%

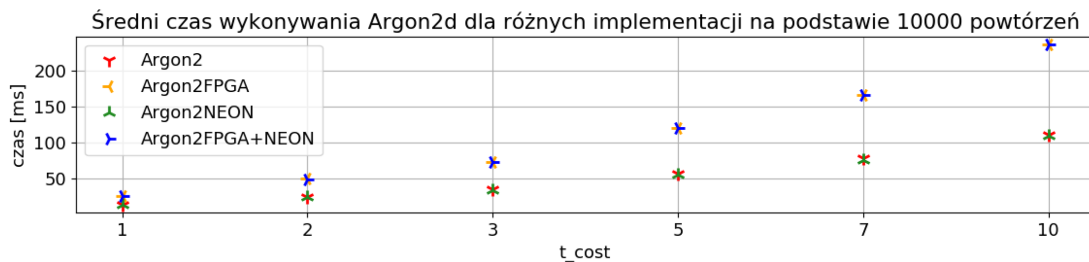
Tabela Z1.2. Porównanie różnych implementacji, parametr `tcost`: 10, liczba powtórzeń: 10000, liczba wywołań `FillBlock`: 23920000, `blake2b_compress`: 2590000.

Funkcja \ Implementacja	ARM (ref)	ARM + FPGA	ARM + Neon	ARM + FPGA + Neon
1 wywołanie `FillBlock`	26,73us	15,88us	31,68us	19,61us
% czasu spędzony w `FillBlock`	81,64%	87,6%	93,64%	81,64%
1 wywołanie `blake2b_compress`	12,08us	11,90us	11,54us	11,66us
% czasu spędzony w `blake2b_compress`	3,99%	7,11%	3,69%	11,44%

W każdym przypadku zaobserwowano pewien wzrost czasu wykonywania funkcji `FillBlock` na skutek zastosowania jednostki *Neon*. Wynika to najprawdopodobniej z dodatkowego czasu potrzebnego na przenoszenie danych pomiędzy różnymi pamięciami.
W przypadku implementacji z koprocesorem, dane są przenoszone z pamięci podręcznej procesora do bufora DMA w pamięci DDR RAM, albo odwrotnie. Zatem wraz z dodaniem instrukcji realizowanej w jednostce *Neon*, dane po drodze przechodzą jeszcze przez tę jednostkę.
W przypadku implementacji bez koprocesora, dane nie są przenoszone do żadnej zewnętrznej pamięci. Procesor pobiera z pamięci podręcznej operandy, a następnie wynik

operacji zapisuje w miejscu jednego z nich. W związku z tym należy przypuszczać, że narzut związany z przenoszeniem danych do pamięci jednostki *Neon* jest większy, niż procentowy zysk wynikający z przyspieszenia obliczeń w oparciu o instrukcję SIMD. Prawdpodobnie musiały by to być minimum 2-3 różne opracje wykonywane kolejno po sobie w jednostce *Neon*, aby faktycznie dało to jakieś przyspieszenie.

Wyniki pomiaru z wykorzystaniem funkcji `gettimeofday` przedstawiono na rys.Z1.1 oraz w tab.Z1.3.



Rysunek Z1.1. Rzeczywisty czas prowadzenia obliczeń w funkcji parametru `tcost`, uśredniony na podstawie 10000 powtórzeń

Tabela Z1.3. Średni czas dla różnych implementacji Argon2d na podstawie 10000 powtórzeń.

Wartość `tcost`	ARM (ref)	ARM + FPGA	ARM + Neon	ARM + FPGA + Neon
1	13,64 ms	26,07 ms	13,41 ms	25,93 ms
2	24,57 ms	49,39 ms	24,09 ms	49,16 ms
3	35,19 ms	72,81 ms	34,22 ms	72,83 ms
5	56,81 ms	119,68 ms	55,49 ms	119,72 ms
7	78,38 ms	166,58 ms	75,87 ms	166,49 ms
10	110,91 ms	236,23 ms	108,72 ms	236,32 ms

Dla implementacji bez koprocesora zaobserwowano nieznaczne, tzn. do około 3% przyspieszenie obliczeń, dzięki wykorzystaniu jednostki *Neon*. Dla implementacji wykorzystujących koprocesor uzyskano zbliżone wyniki niezależnie od sposobu realizacji `XORBlock`.

Bardzo wyraźnie widać rozbieżności w zmierzonym czasie pomiędzy wynikami z *gprof*, a uzyskanymi poprzez bezpośredni pomiar czasu. Dla realizacji na tylko i wyłącznie CPU, z tab.Z1.1 wynika, że w funkcji `FillBlock` procesor spędza 58,7% czasu wykonywania algorytmu, przy 26,68us przypadających na wywołanie tej funkcji i 2320000 wywołaniach funkcji na 10000 iteracji algorytmu. Z tego wynika, że *Argon2d* powinien zajmować około 10,55ms na jedną iterację. Zestawiając to następnie z wynikiem z tab.Z1.3 uzyskuje się około 30% rozbieżności. Dla pomiarów implementacji wykorzystujących koprocesor wyniki różnią się około 4-, 5-krotnie.

Tak znaczące różnice wynikają z odmiennych sposobów dokonywania pomiaru. *Gprof* opiera się na próbkowaniu grafu wywołań. Najczęściej, co 0,01s sprawdza stan grafu.

Próbkowanie jednak może odbywać się tylko i wyłącznie wtedy, kiedy analizowany proces jest w stanie pracującym. W związku z tym, *gprof* może zaniżać czasy. Procentowy udział funkcji w całkowitym czasie wykonania jest mierzony bez uwzględnienia czasu potrzebnego na operacje wejścia/wyjścia czy uśpienie przy pomocy funkcji `sleep` i podobnych. Należy zatem przypuszczać że rozbieżność w pomiarach dla CPU wynika z faktu, że czasami innym procesom był przydzielany czas procesora, a mierzony oczekiwał w stanie gotowości. W przypadku implementacji z koprocesorem można przypuszczać, że mają znaczenie jeszcze 2 kolejne przyczyny:

1. oczekiwanie na operacje wejścia/wyjścia podczas komunikacji z koprocesorem,
2. uśpienie procesu na 1us poprzez wywołanie `usleep(1)` wewnątrz pętli oczekującej na zakończenie pracy koprocesora.

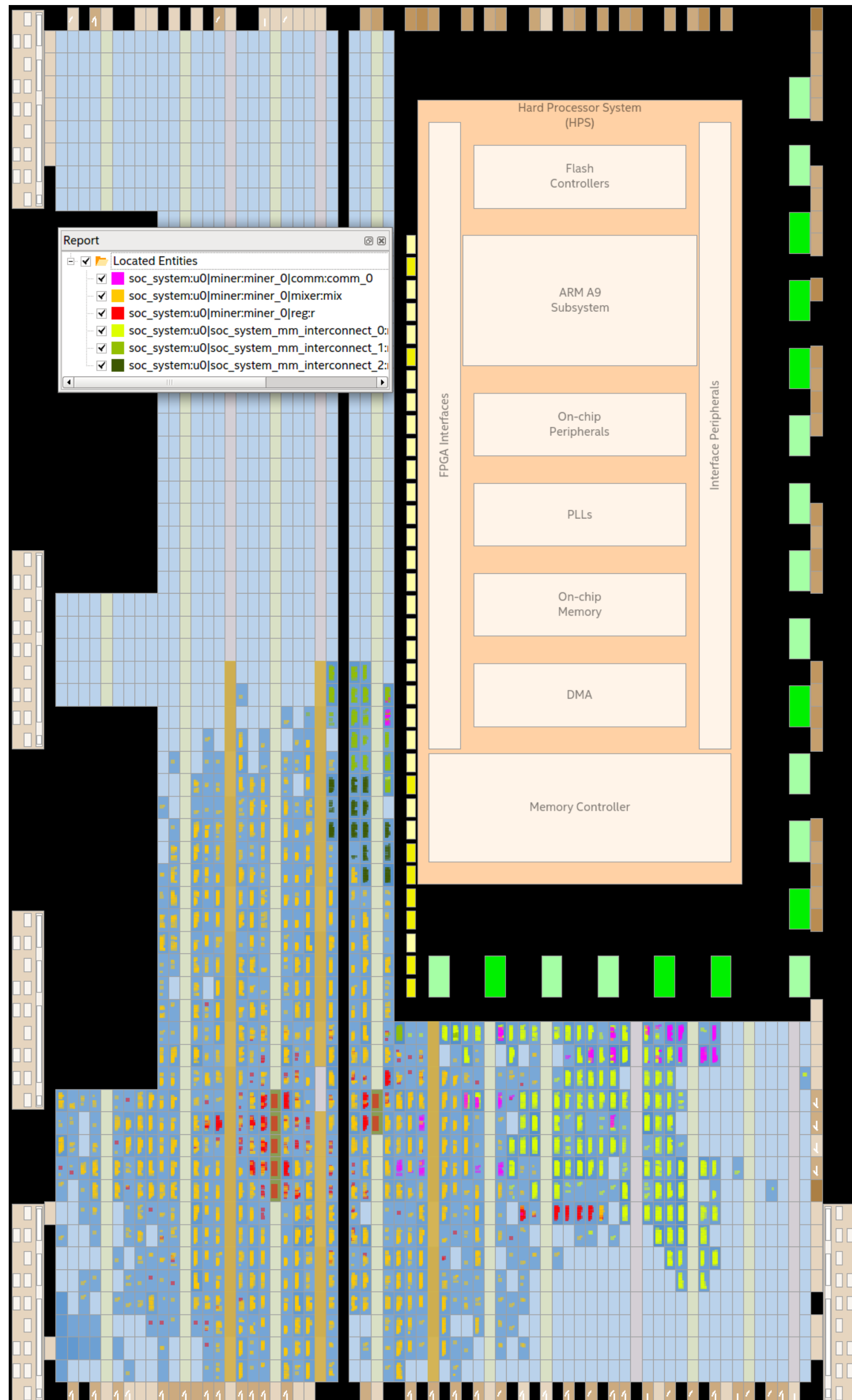
W kodzie programu wygląda to następująco:

```
while(!ioctl(fminer, MINER_IOC_READ_STATUS, NULL)) usleep(1);
```

Obie przyczyny są związane z przeniesieniem procesu do stanu uśpionego, z którego potem przejdzie do stanu gotowości do wykonania. Natomiast po zablokowaniu przez operację wejścia/wyjścia proces powinien mieć znacznie wyższy priorytet w kolejce procesów gotowych, niż po zablokowaniu przez uśpienie.

W celu weryfikacji tego przypuszczenia zaprogramowano w układzie programowalnym, obok koprocesora, analizator logiczny dostępny z wbudowanego w środowisko *Quartus* narzędzia *SignalTap*. Następnie zarejestrowano cały cykl pracy koprocesora. W 6. us pomiaru zauważono żądanie odczytu rejestru stanu procesora. Moment żądania odczytu zaznaczono na rys.6.4 literą E, natomiast odpowiedź koprocesora - F. Przez następne 35us, do końca okresu akwizycji nie zarejestrowano kolejnych żądań. Należy zatem wnioskować, że mierzony proces faktycznie oczekuje w kolejce na przydział czasu procesora.

Załącznik 2. Rozmieszczenie bloków w układzie FPGA



Rysunek Z2.1. Rozmieszczenie bloków po syntezie w układzie programowalnym, widok całego układu. Bloki DSP zaznaczone są szaro-pomarańczowymi kolumnami, pamięci *M10K* - szaro-zielonymi